The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.10.3

COLLABORATORS			
	TITLE : The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.10.3		*REFERENCE* :
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	The GHC Team	May 7, 2009	

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/

glasgow-haskell-bugs: This list is for reporting and discussing GHC bugs. However, please see Section 1.3 before posting here.

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

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

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

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

cvs-ghc: The hardcore GHC developers hang out here. This list also gets commit message from the GHC darcs repository. There are other lists for other darcs repositories (most notably `cvs-libraries`).

list email address: cvs-ghc@haskell.org

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

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

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

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

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 4.10.3) 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 darcs 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 darcs 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 4.5).

1.5 Release notes for version 6.10.3

6.10.3 is a bugfix release over 6.10.2. Very few changes have been make. They are listed below.

- Control-C now works again in GHCi. In 6.10.2 it was ignored on some platforms due to the changes in signal handling.
- GHCi uses Haskeline, rather than editline, for line editing. This fixes a crash when pressing control-C on some platforms, removes a C library dependency, and provides a better user interface.
- It is now possible to build PDF and PS versions of the users guide, using dblatex.

1.6 Release notes for version 6.10.2

6.10.2 is a bugfix release over 6.10.1. Many bugs in the compiler, libraries and build system have been fixed. However, most library APIs have not changed, so code that worked with 6.10.1 should continue to work with 6.10.2. The notable bug fixes are listed below.

- Setting stdin to `NoBuffering` now works on Windows.
- `System.FilePath.dropTrailingPathSeparator "\\"` will now return `"\\"` rather than `""`.
- The compatibility `Control.OldException` functions now catch new-style exceptions as `DynExceptions`.
- It is now possible to create C finalizers which, unlike Haskell finalizers, are guaranteed to run.
- There is now a `+RTS -xm<address>` flag, which tells GHC where to try to allocate memory.
- There is now a `+RTS --machine-readable` flag. Currently it only affects the `+RTS -t` output.
- There are now more fields printed by the `+RTS --info` flag.
- `ghc-pkg` will now complain if told to use a `package.conf` that doesn't exist.
- `ghc-pkg check` now looks for missing files (including .hi files), and reports all packages that are transitively broken.
- The `TypeFamilies` extension now implies the `RelaxedPolyRec` extension.
- We now ship with haddock 2.4.2 (was 2.3.0).
- The version of base has been increased from 4.0.0.0 to 4.1.0.0.
- The version of base(compat) has been increased from 3.0.3.0 to 3.0.3.1.
- The version of Cabal has been increased from 1.6.0.1 to 1.6.0.3.
- The version of containers has been increased from 0.2.0.0 to 0.2.0.1.

- The version of directory has been increased from 1.0.0.2 to 1.0.0.3.
- The version of extensible-exceptions has been increased from 0.1.0.0 to 0.1.0.1.
- The version of filepath has been increased from 1.1.0.1 to 1.1.0.2.
- The version of hpc has been increased from 0.5.0.2 to 0.5.0.3.
- The version of integer-gmp has been increased from 0.1.0.0 to 0.1.0.1.
- The version of old-time has been increased from 1.0.0.1 to 1.0.0.2.
- The version of process has been increased from 1.0.1.0 to 1.0.1.1.
- The version of syb has been increased from 0.1.0.0 to 0.1.0.1.
- The version of template-haskell has been increased from 2.3.0.0 to 2.3.0.1.
- The version of unix has been increased from 2.3.1.0 to 2.4.0.0.

1.7 Release notes for version 6.10.1

The significant changes to the various parts of the compiler are listed in the following sections.

1.7.1 User-visible compiler changes

- The new QuasiQuotes language extension adds general quasi-quotation, as described in "Nice to be Quoted: Quasiquoting for Haskell" (Geoffrey Mainland, Haskell Workshop 2007). See Section 7.9.5 for more information.
- The new ViewPatterns language extension allows "view patterns". The syntax for view patterns is `expression -> pattern` in a pattern. For more information, see Section 7.3.5.
- GHC already supported (e op) postfix operators, but this support was enabled by default. Now you need to use the PostfixOperators language extension if you want it. See Section 7.3.10 for more information on postfix operators.
- The new TransformListComp language extension enables implements generalised list comprehensions, as described in the paper "Comprehensive comprehensions" (Peyton Jones & Wadler, Haskell Workshop 2007). For more information see Section 7.3.8.
- If you want to use impredicative types then you now need to enable the ImpredicativeTypes language extension. See Section 7.8.5 for more information.
- FFI change: header files are now *not used* when compiling via C. The `-#include` flag, the `includes` field in `.cabal` files, and header files specified in a `foreign import` declaration all have no effect when compiling Haskell source code.

  This change has important ramifications if you are calling FFI functions that are defined by macros (or renamed by macros). If you need to call one of these functions, then write a C wrapper for the function and call the wrapper using the FFI instead. In this way, your code will work with GHC 6.10.1, and will also work with `-fasm` in older GHCs.

  This change was made for several reasons. Firstly, `-fvia-C` now behaves consistently with `-fasm`, which is important because we intend to stop compiling via C in the future. Also, we don't need to worry about the interactions between header files, or CPP options necessary to expose certain functions from the system header files (this was becoming quite a headache). We don't need to worry about needing header files when inlining FFI calls across module or package boundaries; calls can now be inlined freely. One downside is that you don't get a warning from the C compiler when you call a function via the FFI at the wrong type.

  Another consequence of this change is that calling *varargs* functions (such as `printf`) via the FFI no longer works. It has never been officially supported (the FFI spec outlaws it), but in GHC 6.10.1 it may now really cause a crash on certain platforms. Again, to call one of these functions use appropriate fixed-argument C wrappers.
- There is a new languages extension PackageImports which allows imports to be qualified with the package they should come from, e.g.

```
import "network" Network.Socket
```

Note that this feature is not intended for general use, it was added for constructing backwards-compatibility packages such as the `base-3.0.3.0` package. See Section 7.3.15 for more details.

- In earlier versions of GHC, the recompilation checker didn't notice changes in other packages meant that recompilation is needed. This is now handled properly, using MD5 checksums of the interface ABIs.
- GHC now treats the Unicode "Letter, Other" class as lowercase letters. This is an arbitrary choice, but better than not allowing them in identifiers at all. This may be revisited by Haskell'.
- In addition to the `DEPRECATED` pragma, you can now attach arbitrary warnings to declarations with the new `WARNING` pragma. See Section 7.13.4 for more details.
- If GHC is failing due to `-Werror`, then it now emits a message telling you so.
- GHC now warns about unrecognised pragmas, as they are often caused by a typo. The `-fwarn-unrecognised-pragmas` controls whether this warning is emitted. The warning is enabled by default.
- There is a new flag `-fwarn-dodgy-foreign-imports` which controls a new warning about FFI delcarations of the form

  ```
  foreign import "f" f :: FunPtr t
  ```

  on the grounds that it is probably meant to be

  ```
  foreign import "&f" f :: FunPtr t
  ```

  The warning is enabled by default.
- External core (output only) is working again.
- There is a new flag `-dsuppress-uniques` that makes GHC's intermediate core easier to read. This flag cannot be used when actually generating code.
- There is a new flag `-dno-debug-output` that suppresses all of the debug information when running a compiler built with the `DEBUG` option.
- A bug in earlier versions of GHC meant that sections didn't always need to be parenthesised, e.g. `(+ 1, 2)` was accepted. This has now been fixed.
- The `-fspec-threshold` flag has been replaced by `-fspec-constr-threshold` and `-fliberate-case-threshold` flags. The thresholds can be disabled by `-fno-spec-constr-threshold` and `-fno-liberate-case-threshold`.
- The new flag `-fsimplifier-phases` controls the number of simplifier phases run during optimisation. These are numbered from n to 1 (by default, n=2). Phase 0 is always run regardless of this flag.
- Simplifier phases can have an arbitrary number of tags assigned to them, and multiple phases can share the same tags. The tags can be used as arguments to the new flag `-ddump-simpl-phases` to specify which phases are to be dumped.

  For example, `-ddump-simpl-phases=main` will dump the output of phases 2, 1 and 0 of the initial simplifier run (they all share the "main" tag) while `-ddump-simpl-phases=main:0` will dump only the output of phase 0 of that run.

  At the moment, the supported tags are main (the main, staged simplifier run (before strictness)), post-worker-wrapper (after the w/w split), post-liberate-case (after LiberateCase), and final (final clean-up run)

  The names are somewhat arbitrary and will change in the future.
- The `-fno-method-sharing` flag is now dynamic (it used to be static).

1.7.2 Deprecated flags

- The new flag `-fwarn-deprecated-flags`, controls whether we warn about deprecated flags and language extensions. The warning is on by default.
- The following language extensions are now marked as deprecated; expect them to be removed in a future release:
  - `RecordPuns` (use `NamedFieldPuns` instead)
  - `PatternSignatures` (use `ScopedTypeVariables` instead)
- The following flags are now marked as deprecated; expect them to be removed in a future release:
  - `-Onot` (use `-O0` instead)
  - `-Wnot` (use `-w` instead)
  - `-frewrite-rules` (use `-fenable-rewrite-rules` instead)
  - `-no-link` (use `-c` instead)
  - `-recomp` (use `-fno-force-recomp` instead)
  - `-no-recomp` (use `-fforce-recomp` instead)
  - `-syslib` (use `-package` instead)
  - `-fth` (use the `TemplateHaskell` language extension instead)
  - `-ffi`, `-fffi` (use the `ForeignFunctionInterface` extension instead)
  - `-farrows` (use the `Arrows` language extension instead)
  - `-fgenerics` (use the `Generics` language extension instead)
  - `-fno-implicit-prelude` (use the `NoImplicitPrelude` language extension instead)
  - `-fbang-patterns` (use the `BangPatterns` language extension instead)
  - `-fno-monomorphism-restriction` (use the `NoMonomorphismRestriction` language extension instead)
  - `-fmono-pat-binds` (use the `MonoPatBinds` language extension instead)
  - `-fextended-default-rules` (use the `ExtendedDefaultRules` language extension instead)
  - `-fimplicit-params` (use the `ImplicitParams` language extension instead)
  - `-fscoped-type-variables` (use the `ScopedTypeVariables` language extension instead)
  - `-fparr` (use the `PArr` language extension instead)
  - `-fallow-overlapping-instances` (use the `OverlappingInstances` language extension instead)
  - `-fallow-undecidable-instances` (use the `UndecidableInstances` language extension instead)
  - `-fallow-incoherent-instances` (use the `IncoherentInstances` language extension instead)
  - `-optdep-s` (use `-dep-suffix` instead)
  - `-optdep-f` (use `-dep-makefile` instead)
  - `-optdep-w` (has no effect)
  - `-optdep--include-prelude` (use `-include-pkg-deps` instead)
  - `-optdep--include-pkg-deps` (use `-include-pkg-deps` instead)
  - `-optdep--exclude-module` (use `-exclude-module` instead)
  - `-optdep-x` (use `-exclude-module` instead)
- The following flags have been removed:
  - `-no-link-chk` (has been a no-op since at least 6.0)
  - `-fruntime-types` (has not been used for years)
  - `-fhardwire-lib-paths` (use `-dynload sysdep`)
- The `-unreg` flag, which was used to build unregisterised code with a registerised compiler, has been removed. Now you need to build an unregisterised compiler if you want to build unregisterised code.

1.7.3 GHC API changes

- There is now a Ghc Monad used to carry around GHC's Session data. This Monad also provides exception handling functions.
- It is now possible to get the raw characters corresponding to each token the lexer outputs, and thus to reconstruct the original file.
- GHCi implicitly brings all exposed modules into scope with qualified module names. There is a new flag `-fimplicit-import-qualified` that controls this behaviour, so other GHC API clients can specify whether or not they want it.
- There is now haddock documentation for much of the GHC API.

1.7.4 GHCi changes

- You can now force GHCi to interpret a module, rather than loading its compiled code, by prepending a * character to its name, e.g.

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

- By default, GHCi will not print bind results, e.g.

```
Prelude> c <- return 'c'
Prelude>
```

  does not print `'c'`. Use `-fprint-bind-result` if you want the old behaviour.
- GHCi now uses editline, rather than readline, for input. This shouldn't affect its behaviour.
- The GHCi prompt history is now saved in `~/.ghc/ghci_history`.
- GHCi now uses libffi to make FFI calls, which means that the FFI now works in GHCi on a much wider range of platforms (all those platforms that libffi supports).

1.7.5 Runtime system changes

- The garbage collector can now use multiple threads in parallel. The new `-gn` RTS flag controls it, e.g. run your program with `+RTS -g2 -RTS` to use 2 threads. The `-g` option is implied by the usual `-N` option, so normally there will be no need to specify it separately, although occasionally it is useful to turn it off with `-g1`.

  Do let us know if you experience strange effects, especially an increase in GC time when using the parallel GC (use `+RTS -s -RTS` to measure GC time). See Section 4.14.3 for more details.
- It is now possible to generate a heap profile without recompiling your program for profiling. Run the program with `+RTS -hT` to generate a basic heap profile, and use **hp2ps** as usual to convert the heap profile into a `.ps` file for viewing. See Section 4.14.5 for more details.
- If the user presses control-C while running a Haskell program then the program gets an asynchronous UserInterrupt exception.
- We now ignore SIGPIPE by default.
- The `-S` and `-s` RTS flags now send their output to stderr, rather than *`prog`*`.stat`, by default.
- The new `-vg` RTS flag provides some RTS trace messages even in the non-debug RTS variants.

1.7.6 runghc

- runghc now uses the compiler that it came with to run the code, rather than the first compiler that it finds on the PATH.
- If the program to run does not have a `.lhs` extension then runghc now treats it as a `.hs` file. In particular, this means that programs without an extension now work.
- `runghc foo` will now work if `foo.hs` or `foo.lhs` exists.
- runghc can now take the code to run from stdin.

1.7.7 ghc-pkg

- ghc-pkg will refuse to unregister a package on which other packages depend, unless the `--force` option is also supplied.
- ghc-pkg now has a `-no-user-package-conf` flag which instructs it to ignore the user's personal package.conf.
- ghc-pkg no longer allows you to register two packages that differ in case only.
- ghc-pkg no longer allows you to register packages which have unversioned dependencies.
- There is a new command `dump` which is similar to `describe '*'`, but in a format that is designed to be parsable by other tools.

1.7.8 Haddock

- Haddock 2 now comes with GHC.

1.7.9 DPH changes

- DPH is now an extralib.
- There is a new flag `-Odph` that sets the flags recommended when using DPH. Currently it is equivalent to `-O2 -fno-method-sharing -fdicts-cheap -fmax-simplifier-iterations20 -fno-spec-constr-threshold`
- There are now flags `-fdph-seq` and `-fdph-par` for selecting which DPH backend to use.
- The `-fflatten` flag has been removed. It never worked and has now been superceded by vectorisation.

1.7.10 Boot Libraries

1.7.10.1 array

- Version number 0.2.0.0 (was 0.1.0.0)

1.7.10.2 base

- Version number 4.0.0.0 (was 3.0.2.0)
- We also ship a base version 3.0.3.0, so legacy code should continue to work.
- The `Show` instance for `Ratio` now puts spaces around the `%`, as required by Haskell 98.
- There is a new module `Control.Category`.
- `>>>` is no longer a method of the `Arrow` class; instead `Category` is a superclass of `Arrow`.
- `pure` is no longer a method of the `Arrow` class; use `arr` instead.

- `Control.Exception` now uses extensible exceptions. The old style of exceptions are still available in `Control.OldException`, but we expect to remove them in a future release.
- There is a new function `System.Exit.exitSuccess :: IO a` analogous to the existing `System.Exit.exitFailure :: IO a`.
- There are new functions `Data.Either.lefts :: [Either a b] -> [a]`, `Data.Either.rights :: [Either a b] -> [b]` and `Data.Either.partitionEithers :: [Either a b] -> ([a], [b])`.
- The new function `Data.List.subsequences :: [a] -> [[a]]` gives all sublists of a list, e.g. `subsequences "abc" == ["","a","b","ab","c","ac","bc","abc"]`.
- The new function `Data.List.permutations :: [a] -> [[a]]` gives all permutations of a list, e.g. `permutations "abc" == ["abc","bac","cba","bca","cab","acb"]`.
- The new functions `Data.Traversable.mapAccumL` and `Data.Traversable.mapAccumR` generalise their `Data.List` counterparts to work on any `Traversable` type.
- The new function `Control.Exception.blocked :: IO Bool` tells you whether or not exceptions are blocked (as controlled by `Control.Exception.(un)block`).
- There is a new function `traceShow :: Show a => a -> b -> b` in `Debug.Trace`.
- The type of `Control.Monad.forever` has been generalised from `Monad m => m a -> m ()` to `Monad m => m a -> m b`.
- The new value `GHC.Exts.maxTupleSize` tells you the largest tuple size that can be used. This is mostly of use in Template Haskell programs.
- `GHC.Exts` now exports `Down(..)`, `groupWith`, `sortWith` and `the` which are used in the desugaring of generalised comprehensions.
- `GHC.Exts` no longer exports the `Integer` internals. If you want them then you need to get them directly from the new `integer` package.
- The new function `GHC.Conc.threadStatus` allows you to ask whether a thread is running, blocked on an MVar, etc.
- The `Data.Generics` hierarchy has been moved to a new package `syb`.
- The `GHC.Prim` and `GHC.PrimopWrappers` modules have been moved into a new `ghc-prim` package.

1.7.10.3 bytestring

- Version number 0.9.0.1.2 (was 0.9.0.1.1)

1.7.10.4 Cabal

- Version number 1.6.0.1 (was 1.2.4.0)
- Many API changes. See the Cabal docs for more information.

1.7.10.5 containers

- Version number 0.2.0.0 (was 0.1.0.2)
- Various result type now use `Maybe` rather than allowing any Monad.

1.7.10.6 directory

- Version number 1.0.0.2 (was 1.0.0.1)
- No longer defines the UNICODE CPP symbol for packages that use it.

1.7.10.7 editline

- This is a new bootlib, version 0.2.1.0.

1.7.10.8 filepath

- Version number 1.1.0.1 (was 1.1.0.0)

1.7.10.9 ghc-prim

- This is a new bootlib, version 0.1.0.0.

1.7.10.10 haskell98

- Version number 1.0.1.0 (unchanged)

1.7.10.11 hpc

- Version number 0.5.0.2 (was 0.5.0.1)

1.7.10.12 integer

- This is a new bootlib, version 0.1.0.0.

1.7.10.13 old-locale

- Version number 1.0.0.1 (was 1.0.0.0)

1.7.10.14 old-time

- Version number 1.0.0.1 (was 1.0.0.0)

1.7.10.15 packedstring

- Version number 0.1.0.1 (was 0.1.0.0)

1.7.10.16 pretty

- Version number 1.0.1.0 (was 1.0.0.0)
- There is a new combinator `zeroWidthText :: String -> Doc` for printing things like ANSI escape sequences.

1.7.10.17 process

- Version number 1.0.1.0 (was 1.0.0.1)
- The `System.Process` API has been overhauled. The new API is a superset of the old API, however.

1.7.10.18 random

- Version number 1.0.0.1 (was 1.0.0.0)

1.7.10.19 readline

- This is no longer a bootlib; editline replaces it.

1.7.10.20 syb

- This is a new bootlib, version 0.1.0.0.

1.7.10.21 template-haskell

- Version number 2.3.0.0 (was 2.2.0.0)
- The datatypes now have support for Word primitives.
- `currentModule :: Q String` has been replaced with `location :: Q Loc`, where `Loc` is a new datatype.

1.7.10.22 unix

- Version number 2.3.1.0 (was 2.3.0.1)
- The `System.Posix.Terminal.BaudRate` type now includes `B57600` and `B115200` constructors.

1.7.10.23 Win32

- Version number 2.2.0.0 (was 2.1.1.1)
- No longer defines the UNICODE CPP symbol for packages that use it.

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 6.8.1: http://www.haskell.org/ghc/  :? for help
Loading package base ... 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. For Hugs users: many things work the same as in Hugs, so you should be able to get going straight away.

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.

2.2 Loading source files

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

```
main = print (fac 20)

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

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

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 2.4.3). 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 4.6.8).

[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\username`.

[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 4.6.3.

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 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.

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 2.4.3). 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 type something of type `IO a` for some `a`, then GHCi *executes* it as an IO-computation.

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

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 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. 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. 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 *`*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.

We're 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.

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`.

The scope is manipulated using the `:module` command. For example, if the current scope is `Prelude`, then we can bring into scope the exports from the module `IO` like so:

```
Prelude> :module +IO
Prelude IO> hPutStrLn stdout "hello\n"
hello
Prelude IO>
```

(Note: you can use `import M` as an alternative to `:module +M`, and `:module` can also be shortened to `:m`). The full syntax of the `:module` command is:

```
: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`, GHCi will assume that you really wanted the `Prelude` and add it in for you (if you don't want the `Prelude`, then ask to remove it with `:m -Prelude`).

The scope is automatically set after a `:load` command, to 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).

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.

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 `".

2.4.3.1 `:module` and `:load`

It might seem that `:module` and `:load` do similar things: you can use both to bring a module into scope. However, there is a clear 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`.
- The set of modules that are currently *in scope* at the prompt. This set is modified by `:module`, and it is also set automatically after `:load`, `:add`, and `:reload`.

You cannot add a module to the scope if it is not loaded. This is why trying to use `:module` to load a new module results in the message "`module M is not loaded`".

2.4.3.2 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`.

2.4.3.3 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"]
```

2.4.4 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.

2.4.5 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 => a` 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 98 Report (Revised)) 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.

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>
```

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:

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

```
*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:

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

[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.

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 4.7).

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. `:stepover` works 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. GHCi remembers the last 50 steps in the history (perhaps in the future we'll make this configurable).

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 `+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 7.8.2) 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 4.8.1) 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 `-l`*`lib`* 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 `-L`*`path`* 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 4.10.7).

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 defined 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.

If the `*` symbol is placed before the module name, then *all* the identifiers in scope in *`module`* are shown; otherwise the list is limited to the exports of *`module`*. 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. If the `!` symbol is appended to the command, data constructors and class methods will be listed individually, otherwise, they will only be listed in the context of their data type or class declaration. The `!`-form also annotates the listing with comments giving possible imports for each group of entries.

```
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, in the scope of `Prelude`, the first group of items from `Data.Maybe` have not been imported (but are available in fully qualified form in the GHCi session - see Section 2.4.3), whereas the second group of items have been imported 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`.

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`.

`: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.

See also Section 10.1.

: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. 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.

: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.

: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`.

: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 2.4.3 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.

:set [*option...*] Sets various options. See Section 2.8 for a list of available options and Section 4.17.10 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, and `%%` is replaced by `%`.

: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.

: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 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 languages Show the currently active language flags.

: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*] Single-step from the last breakpoint. With an expression argument, begins evaluation of the expression with a single-step.

: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` command

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:

+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 `-fglasgow-exts`, you would say:

```
Prelude> :set -fglasgow-exts
```

Any GHC command-line option that is designated as *dynamic* (see the table in Section 4.17), may be set using `:set`. To unset an option, you can set the reverse option:

```
Prelude> :set -fno-glasgow-exts
```

Section 4.17 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.

2.9 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. 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 everytime you start GHCi: eg. if your project uses GHC extensions and CPP, and has source files in three subdirectories A, B and C, you might put the following lines in `.ghci`:

```
:set -fglasgow-exts -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

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.

`-read-dot-ghci` Read `./.ghci` and the other startup files (see above). This is normally the default, but the `-read-dot-ghci` option may be used to override a previous `-ignore-dot-ghci` option.

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 11.2.

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.

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 -- -fglasgow-exts Foo` means runghc won't try to use `glasgow-exts` 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-conf --ghc-arg=foo.conf Main.hs`.

Chapter 4

Using GHC

4.1 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:

4.1.1 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`.

4.1.2 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 uses GHC extensions, it will always need to be compiled with the `-fglasgow-exts` 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 -fglasgow-exts #-}
module X where
...
```

`OPTIONS_GHC` pragmas are only looked for at the top of your source files, upto the first (non-literate,non-empty) line not containing `OPTIONS_GHC`. Multiple `OPTIONS_GHC` pragmas are recognised. Do not put comments before, or on the same line as, the `OPTIONS_GHC` pragma.

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).

4.1.3 Setting options in GHCi

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

4.2 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 4.4.

Dynamic Flags Most non-mode flags fall into this category. A dynamic flag may be used on the command line, in a `GHC_OPTIONS` 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 4.17) 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.

4.3 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.

`.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.

`.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.

4.4 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. The available modes 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 4.4.1.

`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 4.4.2 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. This mode also applies if there is no other mode flag specified on the command line, in which case it means that the specified files should be compiled and then linked to form a program. See Section 4.4.3.

ghc -M Dependency-generation mode. In this mode, GHC can be used to generate dependency information suitable for use in a `Makefile`. See Section 4.6.11.

ghc --mk-dll DLL-creation mode (Windows only). See Section 11.6.1.

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-languages Print the supported language extensions.

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 4.8).

4.4.1 Using ghc --make

When given the `--make` option, 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
```

The command line may contain any number of source file names or module names; 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.

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 4.1.2).

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.

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 4.6.3).

4.4.2 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
```

4.4.3 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 whether a native-code generator is used (producing assembly language) or not (producing C). See Section 4.10.6 for more details.

Note: C pre-processing is optional, the `-cpp` flag turns it on. See Section 4.10.3 for more details.

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

4.4.3.1 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`.

4.5 Help and verbosity options

See also the `--help`, `--version`, `--numeric-version`, and `--print-libdir` modes in Section 4.4.

`-n` Does a dry-run, i.e. GHC goes through all the motions of compiling as normal, but does not actually run any external commands.

`-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.

`-vn` 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).

`-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).

`-Hsize` Set the minimum size of the heap to *`size`*. This option is equivalent to `+RTS -Hsize`, see Section 4.14.3.

`-Rghc-timing` Prints a one-line summary of timing statistics for the GHC run. This option is equivalent to `+RTS -tstderr`, see Section 4.14.3.

4.6 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 7.3.3); 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.

4.6.1 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.

4.6.2 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 4.6.7).

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 4.6.3. 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.

4.6.3 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 4.8) for details.

4.6.4 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.

`-outputdir` *`dir`* The `-outputdir` option is shorthand for the combination of `-odir`, `-hidir`, and `-stubdir`.

`-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 4.6.7).

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.

4.6.5 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 aren't generated when using the native code generator, you may need to use `-fvia-C` to force them to be produced).

`-keep-s-file`, `-keep-s-files` Keep intermediate `.s` files.

`-keep-raw-s-file`, `-keep-raw-s-files` Keep intermediate `.raw-s` files. These are the direct output from the C compiler, before GHC does "assembly mangling" to produce the `.s` file. Again, these are not produced when using the native code generator.

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

4.6.6 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).

4.6.7 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 module being compiled) a "minimal" set of import declarations. You can safely replace all the import declarations in "M.hs" with those found in "M.imports". 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 4.4.

4.6.8 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.

4.6.9 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 4.6.10).
  - 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.
- Type synonym declarations are exactly as in Haskell.
- 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 7.8.3), 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.

4.6.10 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 4.6.9).

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 4.6.11

4.6.11 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 4.6.9 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 4.5.)

`-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.

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

4.6.12 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 7.6.3.1), then GHC has to visit that interface file anyway. Example:

```
module A where
  instance C a => D (T a) where ...
  data T a = ...
```

The instance declaration is only relevant if the type T is in use, and if so, GHC will have visited A's interface file to find T's definition.

The only problem comes when a module contains an instance declaration and GHC has no other reason for visiting the module. Example:

```
module Orphan where
  instance C a => D (T a) where ...
  class C a where ...
```

Here, neither D nor T is declared in module Orphan. We call such modules "orphan modules". GHC identifies orphan modules, and visits the interface file of every orphan module below the module being compiled. This is usually wasted work, but there is no avoiding it. You should therefore do your best to have as few orphan modules as possible.

Functional dependencies complicate matters. Suppose we have:

```
module B where
  instance E T Int where ...
  data T = ...
```

Is this an orphan module? Apparently not, because `T` is declared in the same module. But suppose class `E` had a functional dependency:

```
module Lib where
  class E x y | y -> x where ...
```

Then in some importing module M, the constraint `(E a Int)` should be "improved" by setting `a = T`, *even though there is no explicit mention of `T` in M*. These considerations lead to the following definition of an orphan module:

- An *orphan module* 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.

4.7 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-duplicate-exports`, `-fwarn-missing-fields`, `-fwarn-missing-methods`, and `-fwarn-dodgy-foreign-imports`. The following flags are simple ways to select standard "packages" of warnings:

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

`-Wall`: Turns on all warning options that indicate potentially suspicious code. The warnings that are *not* enabled by `-Wall` are `-fwarn-simple-patterns`, `-fwarn-tabs`, `-fwarn-incomplete-record-updates`, `-fwarn-monomorphism-restriction`, and `-fwarn-implicit-prelude`.

`-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-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-warnings-deprecations`: Causes a warning to be emitted when a module, function or type with a WARNING or DEPRECATED pragma is used. See Section 7.13.4 for more details on the pragmas.

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-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-imports`: Causes a warning to be emitted when a a datatype `T` is imported with all constructors, i.e. `T(..)`, but has been exported abstractly, i.e. `T`.

`-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-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: Similarly for incomplete patterns, 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.

-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-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`.

-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-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 ...`.

-fwarn-orphans: This option causes 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 4.6.12 for details.

-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-simple-patterns: Causes the compiler to warn about lambda-bound patterns that can fail, eg. `\(x:xs)->...`. Normally, these aren't treated as incomplete patterns by `-fwarn-incomplete-patterns`.

"Lambda-bound patterns" includes all places where there is a single pattern, including list comprehensions and do-notation. In these cases, a pattern-match failure is quite legitimate, and triggers filtering (list comprehensions) or the monad `fail` operation (monads). For example:

```
f :: [Maybe a] -> [a]
f xs = [y | Just y <- xs]
```

Switching on `-fwarn-simple-patterns` will elicit warnings about these probably-innocent cases, which is why the flag is off 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 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-unused-binds: Report any function definitions (and local bindings) which are unused. For top-level functions, the warning is only given if the binding is not exported.

A definition is regarded as "used" if (a) it is exported, or (b) it is mentioned in the right hand side of another definition that is used, or (c) the function it defines begins with an underscore. The last case provides a way to suppress unused-binding warnings selectively.

Notice that a variable is reported as unused even if it appears in the right-hand side of another unused binding.

-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
```

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.)

4.8 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 4.8.1.

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).

4.8.1 Using Packages

GHC only knows about packages that are *installed*. To see which packages are installed, use the `ghc-pkg` command:

```
$ ghc-pkg list
/usr/lib/ghc-6.4/package.conf:
    base-1.0, haskell98-1.0, template-haskell-1.0, mtl-1.0, unix-1.0,
    Cabal-1.0, haskell-src-1.0, parsec-1.0, network-1.0,
    QuickCheck-1.0, HUnit-1.1, fgl-1.0, X11-1.1, HGL-3.1, OpenGL-2.0,
    GLUT-2.0, stm-1.0, readline-1.0, (lang-1.0), (concurrent-1.0),
    (posix-1.0), (util-1.0), (data-1.0), (text-1.0), (net-1.0),
    (hssource-1.0), rts-1.0
```

An installed package is either *exposed* or *hidden* by default. Packages hidden by default are listed in parentheses (eg. `(lang-1.0)`) in the output above. 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.

To see which modules are provided by a package use the `ghc-pkg` command (see Section 4.8.6):

```
$ 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, then all other versions will become hidden.

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 4.4), 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
```

-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.

-package-name *foo* Tells GHC the the module being compiled forms part of package *foo*. If this flag is omitted (a very common case) then the default package `main` is assumed.

Note: the argument to `-package-name` should be the full package identifier for the package, that is it should include the version number. For example: `-package mypkg-1.2`.

4.8.2 The main package

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

4.8.3 Consequences of packages

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.

4.8.4 Package Databases

A package database is a file, normally called `package.conf` which contains descriptions of installed packages. GHC usually knows about two package databases:

- The global package database, which comes with your GHC installation.

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

- A package database private to each user. On Unix systems this will be `$HOME/.ghc/arch-os-version/package.conf`, and on Windows it will be something like `C:\Documents~And~Settings\user\ghc`. The `ghc-pkg` tool knows where this file should be located, and will create it if it doesn't exist (see Section 4.8.6).

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: for example, packages in the user database will override those of the same name in the global database.

You can control the loading of package databases using the following GHC options:

`-package-conf` *`file`* Read in the package configuration file *`file`* in addition to the system default file and the user's local file. Packages in additional files read this way will override those in the global and user databases.

`-no-user-package-conf` Prevent loading of the user's local package database.

To create a new package database, just create a new file and put the string '`[]`' in it. Packages can be added to the file using the `ghc-pkg` tool, described in Section 4.8.6.

4.8.4.1 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 user and system package databases are appended, in that order. e.g. 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.

4.8.5 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 4.8.7.

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.

- Versions of the Haskell libraries for use with GHCi may also abe included: GHCi cannot load `.a` files directly, instead it will look for an object file called `HSfoo.o` and load that. On some systems, the `ghc-pkg` tool can automatically build the GHCi version of each library, see Section 4.8.6. To build these libraries by hand from the `.a` archive, it is possible to use GNU **ld** as follows:

```
ld -r --whole-archive -o HSfoo.o libHSfoo.a
```

  (replace `--whole-archive` with `-all_load` on MacOS X)

- When building the package as shared object, GHC wraps out the underlying linker so that the user gets 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 4.10.7.

```
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 4.10.6 on how object files must be prepared for shared object linking.

GHC does not maintain detailed cross-package dependency information. It does remember which modules in other packages the current module depends on, but not which things within those imported things.

To compile a module which is to be part of a new package, use the `-package-name` option (Section 4.8.1). Failure to use the `-package-name` 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.

4.8.6 Package management (the `ghc-pkg` command)

The `ghc-pkg` tool allows packages to be added or removed from a package database. By default, the system-wide package database is modified, but alternatively the user's local package database or another specified file can be used.

To see what package databases are in use, say `ghc-pkg list`. The stack of databases that `ghc-pkg` knows about can be modified using the `GHC_PACKAGE_PATH` environment variable (see Section 4.8.4.1, and using `--package-conf` 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-conf` 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) operate on the list of databases specified by the flags `--user`, `--global`, and `--package-conf`. 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.

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 4.8.7.

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 expose *P* Sets the `exposed` flag for package *P* to `True`.

ghc-pkg check Check consistency of dependencies in the package database, and report packages that have missing dependencies.

ghc-pkg hide *P* Sets the `exposed` 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 4.8.7 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 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.

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`:

`--auto-ghci-libs` Automatically generate the GHCi `.o` version of each `.a` Haskell library, using GNU ld (if that is available). Without this option, `ghc-pkg` will warn if GHCi versions of any Haskell libraries in the package don't exist.

GHCi `.o` libraries don't necessarily have to live in the same directory as the corresponding `.a` library. However, this option will cause the GHCi library to be created in the same directory as the `.a` library.

`-f` *`file`*`,-package-conf` *`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-conf`, `--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`,`--version` Output the `ghc-pkg` version number.

When modifying the package database *`file`*, a copy of the original file is saved in *`file`*`.old`, so in an emergency you can always restore the old settings by copying the old file back again.

4.8.7 `InstalledPackageInfo`: a package specification

A package specification is a Haskell record; in particular, it is the record InstalledPackageInfo in the module Distribution.InstalledPackage 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: 1.0
license: BSD3
copyright:
maintainer: libraries@haskell.org
stability:
homepage:
package-url:
description:
category:
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.Resource,
                 System.Posix.Temp,
                 System.Posix.Terminal,
                 System.Posix.Time,
                 System.Posix.Unistd,
                 System.Posix.User,
                 System.Posix.Signals.Exts
import-dirs: /usr/lib/ghc-6.4/libraries/unix
library-dirs: /usr/lib/ghc-6.4/libraries/unix
hs-libraries: HSunix
extra-libraries: HSunix_cbits, dl
include-dirs: /usr/lib/ghc-6.4/libraries/unix/include
includes: HsUnix.h
depends: base-1.0
```

The full Cabal documentation is still in preparation (at time of writing), so in the meantime 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).

`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` (optinoal 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 (optinoal 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.

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 name list) Packages on which this package depends. This field contains packages with explicit versions are required, except that when submitting a package to `ghc-pkg register`, the versions will be filled in if they are unambiguous.

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.

4.9 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.

4.9.1 `-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`.

`-Ofile <file>`: (NOTE: not supported since GHC 4.x. Please ask if you're interested in this.)

For those who need *absolute* control over *exactly* what options are used (e.g., compiler writers, sometimes :-), a list of options can be put in a file and then slurped in with `-Ofile`.

In that file, comments are of the `#`-to-end-of-line variety; blank lines and most whitespace is ignored.

Please ask if you are baffled and would like an example of `-Ofile`!

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 -fvia-C` (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.

4.9.2 `-f*`: platform-independent flags

These flags turn on and off individual optimisations. They are normally set via the `-O` options described above, and as such, you shouldn't need to set any of them explicitly (indeed, doing so could lead to unexpected results). However, there are one or two that may be of interest:

`-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.

`-fignore-asserts`: 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 7.12). This flag is turned on by `-O`.

`-fno-cse` Turns off the common-sub-expression elimination optimisation. Can be useful if you have some `unsafePerformIO` expressions that you don't want commoned-up.

`-fno-strictness` Turns off the strictness analyser; sometimes it eats too many cycles.

`-fno-full-laziness` Turns off the full laziness optimisation (also known as let-floating). 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.

`-fspec-constr` Turn on call-pattern specialisation.

`-fliberate-case` Turn on the liberate-case transformation.

`-fstatic-argument-transformation` Turn on the static argument transformation.

`-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).

`-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.

`-funbox-strict-fields`: This option causes all constructor fields which are marked strict (i.e. "!") to be unboxed or unpacked if possible. It is equivalent to adding an `UNPACK` pragma to every strict constructor field (see Section 7.13.10).

This option is a bit of a sledgehammer: it might sometimes make things worse. Selectively unboxing fields by using `UNPACK` pragmas might be better.

`-funfolding-creation-threshold=`*`n`*: (Default: 45) 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 that unfolding at 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 next option is more useful:

`-funfolding-use-threshold=`*`n`* (Default: 8) This is the magic cut-off figure for unfolding: 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 (see `-funfolding-con-discount`).

4.10 Options related to a particular phase

4.10.1 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.

-pgmm ***cmd*** Use *cmd* as the mangler.

-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 4.10.7.

4.10.2 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 4.10.4).

-optc ***option*** Pass *option* to the C compiler.

-optm ***option*** Pass *option* to the mangler.

-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 4.10.7.

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 4.14).

4.10.3 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 4.10.2).

-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).

__HASKELL98__ If defined, this means that GHC supports the language defined by the Haskell 98 report.

__HASKELL__=98 In GHC 4.04 and later, the `__HASKELL__` macro is defined as having the value `98`.

__HASKELL1__ If defined to *n*, that means GHC supports the Haskell language defined in the Haskell report version *1.n*. Currently 5. This macro is deprecated, and will probably disappear in future versions.

__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).

__CONCURRENT_HASKELL__ This symbol is defined when pre-processing Haskell (input) and pre-processing C (GHC output). Since GHC from version 4.00 now supports concurrent haskell by default, this symbol is always defined.

__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.).

4.10.3.1 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.

4.10.4 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.

4.10.5 Options affecting the C compiler (if applicable)

If you are compiling with lots of foreign calls, you may need to tell the C compiler about some `#include` files. The Right Way to do this is to add an `INCLUDE` pragma to the top of your source file (Section 7.13.3):

```
{-# INCLUDE <X/Xlib.h> #-}
```

Sometimes this isn't convenient. In those cases there's an equivalent command-line option:

```
% ghc -c '-#include <X/Xlib.h>' Xstuff.lhs
```

4.10.6 Options affecting code generation

-fasm Use GHC's native code generator rather than compiling via C. This will compile faster (up to twice as fast), but may produce code that is slightly slower than compiling via C. `-fasm` is the default.

-fvia-C Compile via C instead of using the native code generator. This is the default on architectures for which GHC doesn't have a native code generator.

-fno-code Omit code generation (and all later phases) altogether. Might be of some use if you just want to see dumps of the intermediate compilation phases.

-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 Mac OS X; it works on PowerPC Linux when using the native code generator (-fasm). It is not quite ready to be used yet for x86 Linux. On Windows, position-independent code is never used, and on PowerPC64 Linux, position-independent code is always used, so the flag is a no-op on those platforms.

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

Note that this option also causes GHC to use shared libraries when linking.

4.10.7 Options affecting linking

GHC has to link your code with various libraries, possibly including: user-supplied, GHC-supplied, and system-supplied (`-lm` math library, for example).

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

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

There's one other gotcha to bear in mind when using external libraries: if the library contains a `main()` function, then this will be 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 4.8), 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/MacOS X 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`.

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

-framework-path*dir* On Darwin/MacOS X 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 switches to shared Haskell libraries for linking. See Section 4.8.5 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 4.10.6

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.

`-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`.
- An module name `A`. GHC assumes that the main function is `A.main`.
- An 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`).

`-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:

- Parallelism on a multiprocessor or multicore machine. See Section 4.12.
  The ability to make a foreign call that does not block all other Haskell threads.
  The ability to invoke foreign exported Haskell functions from multiple OS threads.

With `-threaded`, calls to foreign functions are made using the same OS thread that created the Haskell thread (if it was created by a call to a foreign exported Haskell function), or an arbitrary OS thread otherwise (if the Haskell thread was created by `forkIO`).

More details on the use of "bound threads" in the threaded runtime can be found in the Control.Concurrent module.

`-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 4.10.1) and `-optwindres` (Section 4.10.2).

4.11 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.

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

`-Cs` 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.

4.12 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 7.18 we describe the language features that affect parallelism.

4.12.1 Options for SMP parallelism

In order to make use of multiple CPUs, your program must be linked with the `-threaded` option (see Section 4.10.7). Then, to run a program on multiple CPUs, use the RTS `-N` option:

`-Nx` 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`.

Setting `-N` also has the effect of setting `-g` (the number of OS threads to use for garbage collection) to the same value.

There is no means (currently) by which this value may vary after the program has started.

The following options affect the way the runtime schedules threads on CPUs:

`-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. It is probably only of use if you are explicitly scheduling threads onto CPUs with `GHC.Conc.forkOnIO`.

[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.

-qw Migrate a thread to the current CPU when it is woken up. Normally when a thread is woken up after being blocked it will be scheduled on the CPU it was running on last; this option allows the thread to immediately migrate to the CPU that unblocked it.

The rationale for allowing this eager migration is that it tends to move threads that are communicating with each other onto the same CPU; however there are pathalogical situations where it turns out to be a poor strategy. Depending on the communication pattern in your program, it may or may not be a good idea.

4.12.2 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.

GHC's parallelism support is new and experimental. It may make your program go faster, or it might slow it down - either way, we'd be interested to hear from you.

One significant limitation with the current implementation is that the garbage collector is still single-threaded, and all execution must stop when GC takes place. This can be a significant bottleneck in a parallel program, especially if your program does a lot of GC. If this happens to you, then try reducing the cost of GC by tweaking the GC settings (Section 4.14.3): enlarging the heap or the allocation area size is a good start.

4.13 Platform-specific Flags

Some flags only make sense for particular target platforms.

-monly-[32]-regs: (iX86 machines) GHC tries to "steal" four registers from GCC, for performance reasons; it almost always works. However, when GCC is compiling some modules with four stolen registers, it will crash, probably saying:

```
Foo.hc:533: fixed or forbidden register was spilled.
This may be due to a compiler bug or to impossible asm
statements or clauses.
```

Just give some registers back with `-monly-N-regs`. Try '3' first, then '2'. If '2' doesn't work, please report the bug to us.

4.14 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, profiling, etc.

You have some control over the behaviour of the RTS, by giving special command-line arguments to your program.

When your Haskell program starts up, its RTS extracts command-line arguments bracketed between `+RTS` and `-RTS` as its own. For example:

```
% ./a.out -f +RTS -p -S -RTS -h foo bar
```

The RTS will snaffle `-p -S` for itself, and the remaining arguments `-f -h foo bar` will be handed to your program if/when it calls `System.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 -f` 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 increase the maximum heap size for a compilation to 128M, you would add `+RTS -M128m -RTS` to the command line.

4.14.1 Setting global RTS options

RTS options are also taken from the environment variable `GHCRTS`. For example, to set the maximum heap size to 128M for all GHC-compiled programs (using an `sh`-like shell):

```
GHCRTS='-M128m'
export GHCRTS
```

RTS options taken from the `GHCRTS` environment variable can be overridden by options given on the command line.

4.14.2 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 4.11) 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.

`-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`.

4.14.3 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.

-Asize [Default: 256k] 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).

-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.

-cn [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*.

-Ffactor [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.

-Ggenerations [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).

-gthreads [Default: 1] [new in GHC 6.10] Set the number of threads to use for garbage collection. This option is only accepted when the program was linked with the -threaded option; see Section 4.10.7.

The garbage collector is able to work in parallel when given more than one OS thread. Experiments have shown that this usually results in a performance improvement given 3 cores or more; with 2 cores it may or may not be beneficial, depending on the workload. Bigger heaps work better with parallel GC, so set your -H value high (3 or more times the maximum residency). Look at the timing stats with +RTS -s to see whether you're getting any benefit from parallel GC or not. If you find parallel GC is significantly *slower* (in elapsed time) than sequential GC, please report it as a bug.

This value is set automatically when the -N option is used, so the only reason to use -g would be if you wanted to use a different number of threads for GC than for execution. For example, if your program is strictly single-threaded but you still want to benefit from parallel GC, then it might make sense to use -g rather than -N.

-Hsize [Default: 0] This option provides a "suggested heap size" for the garbage collector. The garbage collector will use about this much memory until the program residency grows and the heap size needs to be expanded to retain reasonable performance.

By default, the heap will start small, and grow and shrink as necessary. This can be bad for performance, so if you have plenty of memory it's worthwhile supplying a big -H*size*. For improving GC performance, using -H*size* is usually a better bet than -A*size*.

-Iseconds (default: 0.3) In the threaded and SMP versions of the RTS (see -threaded, Section 4.10.7), 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.

`-k`*size* [Default: 1k] Set the initial stack size for new threads. Thread stacks (including the main thread's stack) live on the heap, and grow as required. The default value is good for concurrent applications with lots of small threads; if your program doesn't fit this model then increasing this option may help performance.

The main thread is normally started with a slightly larger heap to cut down on unnecessary stack growth while the program is starting up.

`-K`*size* [Default: 8M] Set the maximum stack size for an individual thread to *size* bytes. This option is there purely 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[`*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` 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, 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 bytes allocated by the program. This may be less than the peak memory use, as some may be freed.
- The total number of garbage collections that occurred.
- The average and maximum space used by your program. 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 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

  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. This may be less than the peak memory use, as some may be freed.
- GHC uses a copying garbage collector. "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 packs data into so-called "megablocks".
- 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 used multiple threads, the total CPU time used for garbage collecting that generation, and the total wall clock time elapsed while garbage collecting that generation.
- Next there is the CPU time and wall clock time elapsedm broken down by what the runtiem 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 occured this garbage collection.
- How many page faults occured since the end of the last garbage collection.
- Which generation is being garbage collected.

4.14.4 RTS options for concurrency and parallelism

The RTS options related to concurrency are described in Section 4.11, and those for parallelism in Section 4.12.1.

4.14.5 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 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).

4.14.6 RTS options for hackers, debuggers, and over-interested souls

These RTS options might be used (a) to avoid a GHC bug, (b) to see "what's really happening", or (c) because you feel like it. Not recommended for everyday use!

-B Sound the bell at the start of each (major) garbage collection.

Oddly enough, people really do use this option! Our pal in Durham (England), Paul Callaghan, writes: "Some people here use it for a variety of purposes—honestly!—e.g., confirmation that the code/machine is doing something, infinite loop detection, gauging cost of recently added code. Certain people can even tell what stage [the program] is in by the beep pattern. But the major use is for annoying others in the same office…"

-D*num* An RTS debugging flag; varying quantities of output depending on which bits are set in *num*. Only works if the RTS was compiled with the `DEBUG` option.

-r*file* Produce "ticky-ticky" statistics at the end of the program run. The *file* business works just like on the `-S` RTS option (above).

"Ticky-ticky" statistics are counts of various program actions (updates, enters, etc.) The program must have been compiled using `-ticky` (a.k.a. "ticky-ticky profiling"), and, for it to be really useful, linked with suitable system libraries. Not a trivial undertaking: consult the installation guide on how to set things up for easy "ticky-ticky" profiling. For more information, see Section 5.7.

-xc (Only available when the program is compiled for profiling.) When an exception is raised in the program, this option causes the current cost-centre-stack 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 -auto-all` and running with `+RTS -xc -RTS` will tell you exactly the call stack at the point the error was raised.

The output contains one line for each exception raised in the program (the program might raise and catch several exceptions during its execution), where each line is of the form:

```
< cc1, ..., ccn >
```

each cc_i is a cost centre in the program (see Section 5.1), and the sequence represents the "call stack" at the point the exception was raised. The leftmost item is the innermost function in the call stack, and the rightmost item is the outermost function.

-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.)

4.14.7 "Hooks" to change RTS behaviour

GHC lets you exercise rudimentary control over the 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 all 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.

The hook `ghc_rts_opts`lets you set RTS options permanently for a given program. 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 -K1m`, place the following definition in a C source file:

```
char *ghc_rts_opts = "-H128m -K1m";
```

Compile the C file, and include the object file on the command line when you link your Haskell program.

These flags are interpreted first, before any RTS flags from the `GHCRTS` environment variable and any flags on the command line.

You can also 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.

For examples of the use of these hooks, see GHC's own versions in the file `ghc/compiler/parser/hschooks.c` in a GHC source tree.

4.14.8 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` (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`), `t` (ticky-ticky profiling) 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.)

4.15 Generating and compiling External Core Files

GHC can dump its optimized intermediate code (said to be in "Core" format) to a file as a side-effect of compilation. Non-GHC back-end tools can read and process Core files; these files have the suffix `.hcr`. The Core format is described in An External Representation for the GHC Core Language, and sample tools for manipulating Core files (in Haskell) are in the GHC source distribution directory under `utils/ext-core`. Note that the format of `.hcr` files is *different* from the Core output format that GHC generates for debugging purposes (Section 4.16), though the two formats appear somewhat similar.

The Core format natively supports notes which you can add to your source code using the `CORE` pragma (see Section 7.13).

`-fext-core` Generate `.hcr` files.

Currently (as of version 6.8.2), GHC does not have the ability to read in External Core files as source. If you would like GHC to have this ability, please make your wishes known to the GHC Team.

4.16 Debugging the compiler

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

4.16.1 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`. 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 (including those generated by the specialisation pass)
- **-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-cse:** CSE pass output
- **-ddump-workwrap:** 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-bcos:** byte code compiler output
- **-ddump-foreign:** dump foreign export stubs

-ddump-simpl-phases: Show the output of each run of the simplifier. Used when even `-dverbose-simpl` doesn't cut it.

-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 upto.

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

-ddump-rn-trace Make the renamer be *real* chatty about what it is upto.

-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-simpl -dcore-lint Foo.hs
```

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

-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.

-dsuppress-uniques Suppress the printing of uniques in debugging output. 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`

-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.

-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.

4.16.2 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.

4.16.3 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.)

4.16.4 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.

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.

4.17 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 4.2), and the flag's opposite (if available).

4.17.1 Help and verbosity options

Section 4.5

Flag	Description	Static/Dynamic	Reverse
`-?`	help	mode	-
`-help`	help	mode	-
`-n`	do a dry run	dynamic	-
`-v`	verbose mode (equivalent to `-v3`)	dynamic	-
`-v`*n*	set verbosity level	dynamic	-
`-V`	display GHC version	mode	-
`--supported-languages`	display the supported language extensions	mode	-
`--info`	display information about the compiler	mode	-
`--version`	display GHC version	mode	-
`--numeric-version`	display GHC version (numeric only)	mode	-
`--print-libdir`	display GHC library directory	mode	-
`-ferror-spans`	output full span in error messages	static	-
`-H`*size*	Set the minimum heap size to *size*	static	-
`-Rghc-timing`	Summarise timing stats for GHC (same as `+RTS -tstderr`)	static	-

4.17.2 Which phases to run

Section 4.4.3

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`	Do not link	dynamic	-
`-x` *suffix*	Override default behaviour for source files	static	-

4.17.3 Alternative modes of operation

Section 4.4

Flag	Description	Static/Dynamic	Reverse
`--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 4.4.1 for details..	mode	-
`-e` *expr*	Evaluate *expr*; see Section 4.4.2 for details.	mode	-
`-M`	Generate dependency information suitable for use in a `Makefile`; see Section 4.6.11 for details.	mode	-

4.17.4 Redirecting output

Section 4.6.4

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	-
`-outputdir` *dir*	set output directory	dynamic	-

4.17.5 Keeping intermediate files

Section 4.6.5

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

4.17.6 Temporary files

Section 4.6.6

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

4.17.7 Finding imports

Section 4.6.3

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

4.17.8 Interface file options

Section 4.6.7

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 4.4.		

4.17.9 Recompilation checking

Section 4.6.8

Flag	Description	Static/Dynamic	Reverse
`-fforce-recomp`	Turn off recompilation checking; implied by any `-ddump-X` option	dynamic	`-fno-force-recomp`

4.17.10 Interactive-mode options

Section 2.9

Flag	Description	Static/Dynamic	Reverse
`-ignore-dot-ghci`	Disable reading of `.ghci` files	static	-
`-read-dot-ghci`	Enable reading of `.ghci` files	static	-
`-fbreak-on-exception`	Break on any exception thrown	dynamic	`-fno-break-on-exception`
`-fbreak-on-error`	Break on uncaught exceptions and errors	dynamic	`-fno-break-on-error`
`-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	-

4.17.11 Packages

Section 4.8

Flag	Description	Static/Dynamic	Reverse
`-package-name` *`P`*	Compile to be part of package *`P`*	dynamic	-
`-package` *`P`*	Expose package *`P`*	static/`:set`	-
`-hide-all-packages`	Hide all packages by default	static	-
`-hide-package` *`name`*	Hide package *`P`*	static/`:set`	-
`-ignore-package` *`name`*	Ignore package *`P`*	static/`:set`	-
`-package-conf` *`file`*	Load more packages from *`file`*	static	-
`-no-user-package-conf`	Don't load the user's package config file.	static	-

4.17.12 Language options

Section 7.1

Flag	Description	Static/Dynamic	Reverse
`-fglasgow-exts`	Enable most language extensions	dynamic	`-fno-glasgow-exts`
`-XOverlappingInstances`	Enable overlapping instances	dynamic	`-XNoOverlappingInstances`
`-XIncoherentInstances`	Enable incoherent instances. Implies `-XOverlappingInstances`	dynamic	`-XNoIncoherentInstances`
`-XUndecidableInstances`	Enable undecidable instances	dynamic	`-XNoUndecidableInstances`
`-fcontext-stack=N`*`n`*	set the limit for context reduction. Default is 20.	dynamic	

Flag	Description	Static/Dynamic	Reverse
-XArrows	Enable arrow notation extension	dynamic	-XNoArrows
-XDisambiguateRecordFields	Enable record field disambiguation	dynamic	-XNoDisambiguateRecordFields
-XForeignFunctionInterface	Enable foreign function interface (implied by -fglasgow-exts)	dynamic	-XNoForeignFunctionInterface
-XGenerics	Enable generic classes	dynamic	-XNoGenerics
-XImplicitParams	Enable Implicit Parameters. Implied by -fglasgow-exts.	dynamic	-XNoImplicitParams
-firrefutable-tuples	Make tuple pattern matching irrefutable	dynamic	-fno-irrefutable--tuples
-XNoImplicitPrelude	Don't implicitly import Prelude	dynamic	-XImplicitPrelude
-XNoMonomorphismRestriction	Disable the monomorphism restriction	dynamic	-XMonomorphismRrestriction
-XNoMonoPatBinds	Make pattern bindings polymorphic	dynamic	-XMonoPatBinds
-XRelaxedPolyRec	Relaxed checking for mutually-recursive polymorphic functions	dynamic	-XNoRelaxedPolyRec
-XExtendedDefaultRules	Use GHCi's extended default rules in a normal module	dynamic	-XNoExtendedDefaultRules
-XOverloadedStrings	Enable overloaded string literals.	dynamic	-XNoOverloadedStrings
-XGADTs	Enable generalised algebraic data types.	dynamic	-XNoGADTs
-XTypeFamilies	Enable type families.	dynamic	-XNoTypeFamilies
-XScopedTypeVariables	Enable lexically-scoped type variables. Implied by -fglasgow-exts.	dynamic	-XNoScopedTypeVariables
-XTemplateHaskell	Enable Template Haskell. No longer implied by -fglasgow-exts.	dynamic	-XNoTemplateHaskell
-XQuasiQuotes	Enable quasiquotation.	dynamic	-XNoQuasiQuotes
-XBangPatterns	Enable bang patterns.	dynamic	-XNoBangPatterns
-XCPP	Enable the C preprocessor.	dynamic	-XNoCPP
-XPatternGuards	Enable pattern guards.	dynamic	-XNoPatternGuards
-XViewPatterns	Enable view patterns.	dynamic	-XNoViewPatterns
-XUnicodeSyntax	Enable unicode syntax.	dynamic	-XNoUnicodeSyntax
-XMagicHash	Allow "#" as a postfix modifier on identifiers.	dynamic	-XNoMagicHash
-XNewQualifiedOperators	Enable new qualified operator syntax	dynamic	-XNoNewQualifiedOperators
-XPolymorphicComponents	Enable polymorphic components for data constructors.	dynamic	-XNoPolymorphicComponents
-XRank2Types	Enable rank-2 types.	dynamic	-XNoRank2Types
-XRankNTypes	Enable rank-N types.	dynamic	-XNoRankNTypes
-XImpredicativeTypes	Enable impredicative types.	dynamic	-XNoImpredicativeTypes
-XExistentialQuantification	Enable existential quantification.	dynamic	-XNoExistentialQuantification
-XKindSignatures	Enable kind signatures.	dynamic	-XNoKindSignatures

Flag	Description	Static/Dynamic	Reverse
-XEmptyDataDecls	Enable empty data declarations.	dynamic	-XNoEmptyDataDecls
-XParallelListComp	Enable parallel list comprehensions.	dynamic	-XNoParallelListComp
-XTransformListComp	Enable transform list comprehensions.	dynamic	-XNoTransformListComp
-XUnliftedFFITypes	Enable unlifted FFI types.	dynamic	-XNoUnliftedFFITypes
-XLiberalTypeSynonyms	Enable liberalised type synonyms.	dynamic	-XNoLiberalTypeSynonyms
-XTypeOperators	Enable type operators.	dynamic	-XNoTypeOperators
-XRecursiveDo	Enable recursive do (mdo) notation.	dynamic	-XNoRecursiveDo
-XPArr	Enable parallel arrays.	dynamic	-XNoPArr
-XRecordWildCards	Enable record wildcards.	dynamic	-XNoRecordWildCards
-XNamedFieldPuns	Enable record puns.	dynamic	-XNoNamedFieldPuns
-XDisambiguateRecordFields	Enable record field disambiguation.	dynamic	-XNoDisambiguateRecordFields
-XUnboxedTuples	Enable unboxed tuples.	dynamic	-XNoUnboxedTuples
-XStandaloneDeriving	Enable standalone deriving.	dynamic	-XNoStandaloneDeriving
-XDeriveDataTypeable	Enable deriving for the Data and Typeable classes.	dynamic	-XNoDeriveDataTypeable
-XGeneralizedNewtypeDeriving	Enable newtype deriving.	dynamic	-XNoGeneralizedNewtypeDeriving
-XTypeSynonymInstances	Enable type synonyms.	dynamic	-XNoTypeSynonymInstances
-XFlexibleContexts	Enable flexible contexts.	dynamic	-XNoFlexibleContexts
-XFlexibleInstances	Enable flexible instances.	dynamic	-XNoFlexibleInstances
-XConstrainedClassMethods	Enable constrained class methods.	dynamic	-XNoConstrainedClassMethods
-XMultiParamTypeClasses	Enable multi parameter type classes.	dynamic	-XNoMultiParamTypeClasses
-XFunctionalDependencies	Enable functional dependencies.	dynamic	-XNoFunctionalDependencies
-XPackageImports	Enable package-qualified imports.	dynamic	-XNoPackageImports

4.17.13 Warnings

Section 4.7

Flag	Description	Static/Dynamic	Reverse
-W	enable normal warnings	dynamic	-w
-w	disable all warnings	dynamic	-
-Wall	enable almost all warnings (details in Section 4.7)	dynamic	-w
-Werror	make warnings fatal	dynamic	-Wwarn
-Wwarn	make warnings non-fatal	dynamic	-Werror
-fwarn-unrecognised-pragmas	warn about uses of pragmas that GHC doesn't recognise	dynamic	-fno-warn-unrecognised-pragmas

Flag	Description	Static/Dynamic	Reverse
-fwarn-warnings-deprecations	warn about uses of functions & types that have warnings or deprecated pragmas	dynamic	-fno-warn-warnings-deprecations
-fwarn-deprecated-flags	warn about uses of commandline flags that are deprecated	dynamic	-fno-warn-deprecated-flags
-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-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-record-updates	warn when a record update could fail	dynamic	-fno-warn-incomplete-record-updates
-fwarn-missing-fields	warn when fields of a record are uninitialised	dynamic	-fno-warn-missing-fields
-fwarn-missing-methods	warn when class methods are undefined	dynamic	-fno-warn-missing-methods
-fwarn-missing-signatures	warn about top-level functions without signatures	dynamic	-fno-warn-missing-signatures
-fwarn-name-shadowing	warn when names are shadowed	dynamic	-fno-warn-name-shadowing
-fwarn-orphans	warn when the module contains orphan instance declarations or rewrite rules	dynamic	-fno-warn-orphans
-fwarn-overlapping-patterns	warn about overlapping patterns	dynamic	-fno-warn-overlapping-patterns
-fwarn-simple-patterns	warn about lambda-patterns that can fail	dynamic	-fno-warn-simple-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-monomorphism-restriction	warn when the Monomorphism Restriction is applied	dynamic	-fno-warn-monomorphism-restriction
-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

4.17.14 Optimisation levels

Section 4.9

Flag	Description	Static/Dynamic	Reverse
-O	Enable default optimisation (level 1)	dynamic	-O0
-O*n*	Set optimisation level *n*	dynamic	-O0

4.17.15 Individual optimisations

Section 4.9.2

Flag	Description	Static/Dynamic	Reverse
-fcase-merge	Enable case-merging. Implied by -O.	dynamic	-fno-case-merge
-fdicts-strict	Make dictionaries strict	static	-fno-dicts-strict
-fmethod-sharing	Share specialisations of overloaded functions (default)	dynamic	-fno-method-sharing
-fdo-eta-reduction	Enable eta-reduction. Implied by -O.	dynamic	-fno-do-eta-reduction
-fdo-lambda-eta-expansion	Enable lambda eta-reduction	dynamic	-fno-do-lambda-eta-expansion
-fexcess-precision	Enable excess intermediate precision	dynamic	-fno-excess-precision
-fignore-asserts	Ignore assertions in the source	dynamic	-fno-ignore-asserts
-fignore-interface-pragmas	Ignore pragmas in interface files	dynamic	-fno-ignore-interface-pragmas
-fomit-interface-pragmas	Don't generate interface pragmas	dynamic	-fno-omit-interface-pragmas
-fmax-worker-args	If a worker has that many arguments, none will be unpacked anymore (default: 10)	static	-
-fsimplifier-phases	Set the number of phases for the simplifier (default 2). Ignored with -O0.	dynamic	-
-fmax-simplifier-iterations	Set the max iterations for the simplifier	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	-
-fcse	Turn on common sub-expression elimination. Implied by -O.	dynamic	-fno-cse
-ffull-laziness	Turn on full laziness (floating bindings outwards). Implied by -O.	dynamic	-fno-full-laziness
-frewrite-rules	Switch on all rewrite rules (including rules generated by automatic specialisation of overloaded functions). Implied by -O.	dynamic	-fno-rewrite-rules
-fstrictness	Turn on strictness analysis. Implied by -O.	dynamic	-fno-strictness
-fspec-constr	Turn on the SpecConstr transformation. Implied by -O2.	dynamic	-fno-spec-constr

Flag	Description	Static/Dynamic	Reverse
`-fspec-constr-threshold=n`	Set the size threshold for the SpecConstr transformation to *n* (default: 200)	static	`-fno-spec-constr-threshold`
`-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	static	`-fno-spec-constr-count`
`-fliberate-case`	Turn on the liberate-case transformation. Implied by `-O2`.	dynamic	-fno-liberate-case
`-fstatic-argument-transformation`	Turn on the static argument transformation. Implied by `-O2`.	dynamic	-fno-static-argument-transformation
`-fliberate-case-threshold=n`	Set the size threshold for the liberate-case transformation to *n* (default: 200)	static	`-fno-liberate-case-threshold`
`-funbox-strict-fields`	Flatten strict constructor fields	dynamic	`-fno-unbox-strict-fields`
`-funfolding-creation-threshold`	Tweak unfolding settings	static	`-fno-unfolding-creation-threshold`
`-funfolding-fun-discount`	Tweak unfolding settings	static	`-fno-unfolding-fun-discount`
`-funfolding-keeness-factor`	Tweak unfolding settings	static	`-fno-unfolding-keeness-factor`
`-funfolding-use-threshold`	Tweak unfolding settings	static	`-fno-unfolding-use-threshold`
`-fno-pre-inlining`	Turn off pre-inlining	static	-

4.17.16 Profiling options

Chapter 5

Flag	Description	Static/Dynamic	Reverse
`-auto`	Auto-add `_scc_`s to all exported functions	static	`-no-auto`
`-auto-all`	Auto-add `_scc_`s to all top-level functions	static	`-no-auto-all`
`-caf-all`	Auto-add `_scc_`s to all CAFs	static	`-no-caf-all`
`-prof`	Turn on profiling	static	-
`-ticky`	Turn on ticky-ticky profiling	static	-

4.17.17 Program coverage options

Section 5.6

Flag	Description	Static/Dynamic	Reverse
`-fhpc`	Turn on Haskell program coverage instrumentation	static	-

Flag	Description	Static/Dynamic	Reverse
`-hpcdir dir`	Directory to deposit .mix files during compilation (default is .hpc)	dynamic	–

4.17.18 Haskell pre-processor options

Section 4.10.4

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

4.17.19 C pre-processor options

Section 4.10.3

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

4.17.20 C compiler options

Section 4.10.5

Flag	Description	Static/Dynamic	Reverse
`-#include` *`file`*	Include *`file`* when compiling the `.hc` file	dynamic	-

4.17.21 Code generation options

Section 4.10.6

Flag	Description	Static/Dynamic	Reverse
`-fasm`	Use the native code generator	dynamic	-fvia-C
`-fvia-C`	Compile via C	dynamic	-fasm
`-fno-code`	Omit code generation	dynamic	-
`-fbyte-code`	Generate byte-code	dynamic	-
`-fobject-code`	Generate object code	dynamic	-

4.17.22 Linking options

Section 4.10.7

Flag	Description	Static/Dynamic	Reverse
`-fPIC`	Generate position-independent code (where available)	static	-
`-dynamic`	Use dynamic Haskell libraries (if available)	static	-
`-framework` *name*	On Darwin/MacOS X only, link in the framework *name*. This option corresponds to the `-framework` option for Apple's Linker.	dynamic	-
`-framework-path` *name*	On Darwin/MacOS X 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	-
`-no-link`	Omit linking	dynamic	-
`-split-objs`	Split objects (for libraries)	dynamic	-
`-static`	Use static Haskell libraries	static	-
`-threaded`	Use the threaded runtime	static	-
`-debug`	Use the debugging runtime	static	-
`-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	-

4.17.23 Replacing phases

Section 4.10.1

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	-
`-pgmm` *cmd*	Use *cmd* as the mangler	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	-

Flag	Description	Static/Dynamic	Reverse
`-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	-

4.17.24 Forcing options to particular phases

Section 4.10.2

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	-
`-optm` *option*	pass *option* to the mangler	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	-

4.17.25 Platform-specific options

Section 4.13

Flag	Description	Static/Dynamic	Reverse
`-monly-[432]-regs`	(x86 only) give some registers back to the C compiler	dynamic	-

4.17.26 External core file options

Section 4.15

Flag	Description	Static/Dynamic	Reverse
`-fext-core`	Generate `.hcr` external Core files	static	-

4.17.27 Compiler debugging options

Section 4.16

Flag	Description	Static/Dynamic	Reverse
`-dcore-lint`	Turn on internal sanity checking	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-ddump-asm`	Dump assembly	dynamic	-
`-ddump-bcos`	Dump interpreter byte code	dynamic	-
`-ddump-cmm`	Dump C-- output	dynamic	-
`-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-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-rules`	Dump rules	dynamic	-
`-ddump-simpl`	Dump final simplifier output	dynamic	-
`-ddump-simpl-phases`	Dump output from each simplifier phase	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-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-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	-
`-dsuppress-uniques`	Suppress the printing of uniques in debug output (easier to use **diff**.	static	-
`-dppr-noprags`	Don't output pragma info in dumps	static	-
`-dppr-user-length`	Set the depth for printing expressions in error msgs	static	-

Flag	Description	Static/Dynamic	Reverse
`-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	-

4.17.28 Misc compiler options

Flag	Description	Static/Dynamic	Reverse
`-fno-hi-version-check`	Don't complain about `.hi` file mismatches	static	-
`-dno-black-holing`	Turn off black holing (probably doesn't work)	static	-
`-fhistory-size`	Set simplification history size	static	-
`-funregisterised`	Unregisterised compilation (use `-unreg` instead)	static	-
`-fno-asm-mangling`	Turn off assembly mangling (use `-unreg` instead)	dynamic	-

Chapter 5

Profiling

Glasgow Haskell comes with a time and space profiling system. Its purpose is to help you improve your understanding of your program's execution behaviour, so you can improve it.

Any comments, suggestions and/or improvements you have are welcome. Recommended "profiling tricks" would be especially cool!

Profiling a program is a three-step process:

1. Re-compile your program for profiling with the `-prof` option, and probably one of the `-auto` or `-auto-all` options. These options are described in more detail in Section 5.2
2. Run your program with one of the profiling options, eg. `+RTS -p -RTS`. This generates a file of profiling information. Note that multi-processor execution (e.g. `+RTS -N2`) is not supported while profiling.
3. Examine the generated profiling information, using one of GHC's profiling tools. The tool to use will depend on the kind of profiling information generated.

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 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-graph of cost attributions.

Let's take a look at an example:

```
main = print (nfib 25)
nfib n = if n < 2 then 1 else nfib (n-1) + nfib (n-2)
```

Compile and run this program as follows:

```
$ ghc -prof -auto-all -o Main 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:

```
          Fri May 12 14:06 2000 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -RTS
```

```
        total time  =        0.14 secs   (7 ticks @ 20 ms)
        total alloc =   8,741,204 bytes  (excludes profiling overheads)

COST CENTRE          MODULE     %time %alloc

nfib                 Main       100.0  100.0


                                          individual     inherited
COST CENTRE              MODULE      entries %time %alloc   %time %alloc

MAIN                     MAIN              0    0.0   0.0    100.0 100.0
 main                    Main              0    0.0   0.0      0.0   0.0
 CAF                     PrelHandle        3    0.0   0.0      0.0   0.0
 CAF                     PrelAddr          1    0.0   0.0      0.0   0.0
 CAF                     Main              6    0.0   0.0    100.0 100.0
  main                   Main              1    0.0   0.0    100.0 100.0
   nfib                  Main         242785  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 shortly).

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 `nfib`, 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-graph profile of the program. In the example above, it is clear that the costly call to `nfib` 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 25 + g 25)
f n  = nfib n
g n  = nfib (n `div` 2)
nfib n = if n < 2 then 1 else nfib (n-1) + nfib (n-2)
```

Compile and run this program as before, and take a look at the new profiling results:

```
COST CENTRE              MODULE         scc  %time %alloc   %time %alloc

MAIN                     MAIN             0    0.0   0.0    100.0 100.0
 main                    Main             0    0.0   0.0      0.0   0.0
 CAF                     PrelHandle       3    0.0   0.0      0.0   0.0
 CAF                     PrelAddr         1    0.0   0.0      0.0   0.0
 CAF                     Main             9    0.0   0.0    100.0 100.0
  main                   Main             1    0.0   0.0    100.0 100.0
   g                     Main             1    0.0   0.0      0.0   0.2
    nfib                 Main           465    0.0   0.2      0.0   0.2
   f                     Main             1    0.0   0.0    100.0  99.8
    nfib                 Main        242785  100.0  99.8    100.0  99.8
```

Now although we had two calls to `nfib` in the program, it is immediately clear that it was the call from `f` which took all the time.

The actual meaning of the various columns in the output is:

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

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

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 graph.

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.

5.1.1 Inserting cost centres by hand

Cost centres are just program annotations. When you say `-auto-all` to the compiler, it automatically inserts a cost centre annotation around every top-level function in your program, but you are entirely free to add the cost centre annotations yourself.

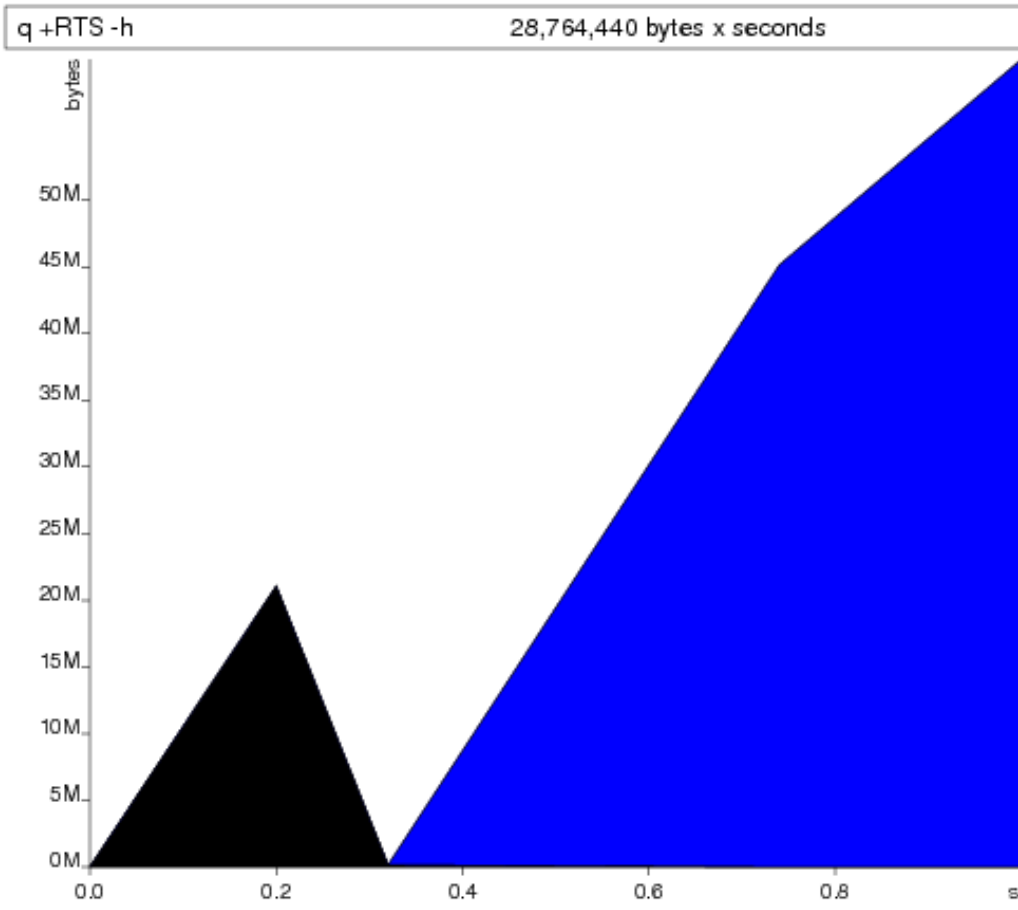
The syntax of a cost centre annotation is

```
{-# 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").

Here is an example of a program with a couple of SCCs:

```
main :: IO ()
main = do let xs = {-# SCC "X" #-} [1..1000000]
          let ys = {-# SCC "Y" #-} [1..2000000]
          print $ last xs
          print $ last $ init xs
          print $ last ys
          print $ last $ init ys
```



which gives this heap profile when run:

5.1.2 Rules for attributing costs

The cost of evaluating any expression in your program is attributed to a cost-centre stack using the following rules:

- If the expression is part of the *one-off* costs of evaluating the enclosing top-level definition, then costs are attributed to the stack of lexically enclosing `SCC` annotations on top of the special `CAF` cost-centre.
- Otherwise, costs are attributed to the stack of lexically-enclosing `SCC` annotations, appended to the cost-centre stack in effect at the *call site* of the current top-level definition[1]. Notice that this is a recursive definition.
- Time spent in foreign code (see Chapter 8) is always attributed to the cost centre in force at the Haskell call-site of the foreign function.

What do we mean by one-off costs? Well, Haskell is a lazy language, and certain expressions are only ever evaluated once. For example, if we write:

```
x = nfib 25
```

then `x` will only be evaluated once (if at all), and subsequent demands for `x` will immediately get to see the cached result. The definition `x` is called a CAF (Constant Applicative Form), because it has no arguments.

For the purposes of profiling, we say that the expression `nfib 25` belongs to the one-off costs of evaluating `x`.

Since one-off costs aren't strictly speaking part of the call-graph of the program, they are attributed to a special top-level cost centre, `CAF`. There may be one `CAF` cost centre for each module (the default), or one for each top-level definition with any one-off costs (this behaviour can be selected by giving GHC the `-caf-all` flag).

If you think you have a weird profile, or the call-graph doesn't look like you expect it to, feel free to send it (and your program) to us at glasgow-haskell-bugs@haskell.org.

[1] The call-site is just the place in the source code which mentions the particular function or variable.

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.

-auto: GHC will automatically add `_scc_` constructs for all top-level, exported functions.

-auto-all: *All* top-level functions, exported or not, will be automatically `_scc_`'d.

-caf-all: 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...

-ignore-scc: Ignore any `_scc_` constructs, so a module which already has `_scc_`s can be compiled for profiling with the annotations ignored.

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: 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.)

-xc This option makes use of the extra information maintained by the cost-centre-stack profiler to provide useful information about the location of runtime errors. See Section 4.14.6.

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 longer run-times 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. `-hc` 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.

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 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 on a 1/50 second frequency.

-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" 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, or an unevaluated closure (thunk).

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-Willhem 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 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.

5.6.1 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
```

The HPC instrumentation is enabled using the -fhpc flag.

```
$ ghc -fhpc Recip.hs --make
```

HPC index (.mix) files are placed placed in .hpc subdirectory. These can be considered like the .hi files for HPC.

```
$ ./Recip
1/3
= 0.(3)
```

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

5.6.2 Options for instrumenting code for coverage

Turning on code coverage is easy, use the -fhpc flag. Instrumented and non-instrumented can be freely mixed. When compiling the Main module GHC automatically detects when there is an hpc compiled file, and adds the correct initialization code.

5.6.3 The hpc toolkit

The hpc toolkit uses a cvs/svn/darcs-like interface, where a single binary contains many function units.

```
$ 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 .tix file after an instrumented binary has generated it, which hpc acting as a conduit between the raw .tix file, and the more detailed reports produced.

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.

5.6.3.1 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                  sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --xml-output                  show output in XML
```

5.6.3.2 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                  sub-directory that contains .mix files
                                  default .hpc [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
```

5.6.3.3 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)
```

5.6.3.4 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)
```

5.6.3.5 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)
```

5.6.3.6 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   sub-directory that contains .mix files
                   default .hpc [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                sub-directory that contains .mix files
                                default .hpc [rarely used]
    --output=FILE               output FILE
```

5.6.4 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.

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

(ToDo: document properly.)

It is possible to compile Glasgow Haskell programs so that they will count lots and lots of interesting things, e.g., number of updates, number of data constructors entered, etc., etc. We call this "ticky-ticky" profiling, because that's the sound a Sun4 makes when it is running up all those counters (*slowly*).

Ticky-ticky profiling is mainly intended for implementors; it is quite separate from the main "cost-centre" profiling system, intended for all users everywhere.

To be able to use ticky-ticky profiling, you will need to have built the ticky RTS. (This should be described in the building guide, but amounts to building the RTS with way "t" enabled.)

To get your compiled program to spit out the ticky-ticky numbers, use a `-r` RTS option. See Section 4.14.

Compiling your program with the `-ticky` switch yields an executable that performs these counts. Here is a sample ticky-ticky statistics file, generated by the invocation **foo +RTS -rfoo.ticky**.

```
 foo +RTS -rfoo.ticky


ALLOCATIONS: 3964631 (11330900 words total: 3999476 admin, 6098829 goods, 1232595 slop)
                                 total words:        2     3     4     5    6+
  69647 (  1.8%) function values                 50.0  50.0   0.0   0.0   0.0
2382937 ( 60.1%) thunks                           0.0  83.9  16.1   0.0   0.0
1477218 ( 37.3%) data values                     66.8  33.2   0.0   0.0   0.0
      0 (  0.0%) big tuples
      2 (  0.0%) black holes                      0.0 100.0   0.0   0.0   0.0
      0 (  0.0%) prim things
  34825 (  0.9%) partial applications             0.0   0.0   0.0 100.0   0.0
      2 (  0.0%) thread state objects             0.0   0.0   0.0   0.0 100.0

Total storage-manager allocations: 3647137 (11882004 words)
        [551104 words lost to speculative heap-checks]

STACK USAGE:

ENTERS: 9400092  of which 2005772 (21.3%) direct to the entry code
                  [the rest indirected via Node's info ptr]
1860318 ( 19.8%) thunks
3733184 ( 39.7%) data values
3149544 ( 33.5%) function values
                  [of which 1999880 (63.5%) bypassed arg-satisfaction chk]
 348140 (  3.7%) partial applications
 308906 (  3.3%) normal indirections
      0 (  0.0%) permanent indirections
```

```
RETURNS: 5870443
2137257 ( 36.4%) from entering a new constructor
                  [the rest from entering an existing constructor]
2349219 ( 40.0%) vectored [the rest unvectored]

RET_NEW:         2137257:  32.5% 46.2% 21.3%  0.0%  0.0%  0.0%  0.0%  0.0%  0.0%
RET_OLD:         3733184:   2.8% 67.9% 29.3%  0.0%  0.0%  0.0%  0.0%  0.0%  0.0%
RET_UNBOXED_TUP:       2:   0.0%  0.0%100.0%  0.0%  0.0%  0.0%  0.0%  0.0%  0.0%

RET_VEC_RETURN : 2349219:   0.0%  0.0%100.0%  0.0%  0.0%  0.0%  0.0%  0.0%  0.0%

UPDATE FRAMES: 2241725 (0 omitted from thunks)
SEQ FRAMES:    1
CATCH FRAMES:  1
UPDATES: 2241725
      0 (  0.0%) data values
  34827 (  1.6%) partial applications
                  [2 in place, 34825 allocated new space]
2206898 ( 98.4%) updates to existing heap objects (46 by squeezing)
UPD_CON_IN_NEW:         0:       0      0      0      0      0      0      0      0      0
UPD_PAP_IN_NEW:     34825:       0      0      0  34825      0      0      0      0      0

NEW GEN UPDATES: 2274700 ( 99.9%)

OLD GEN UPDATES: 1852 (  0.1%)

Total bytes copied during GC: 190096

***********************************************
3647137 ALLOC_HEAP_ctr
11882004 ALLOC_HEAP_tot
  69647 ALLOC_FUN_ctr
  69647 ALLOC_FUN_adm
  69644 ALLOC_FUN_gds
  34819 ALLOC_FUN_slp
  34831 ALLOC_FUN_hst_0
  34816 ALLOC_FUN_hst_1
      0 ALLOC_FUN_hst_2
      0 ALLOC_FUN_hst_3
      0 ALLOC_FUN_hst_4
2382937 ALLOC_UP_THK_ctr
      0 ALLOC_SE_THK_ctr
 308906 ENT_IND_ctr
      0 E!NT_PERM_IND_ctr requires +RTS -Z
[... lots more info omitted ...]
      0 GC_SEL_ABANDONED_ctr
      0 GC_SEL_MINOR_ctr
      0 GC_SEL_MAJOR_ctr
      0 GC_FAILED_PROMOTION_ctr
  47524 GC_WORDS_COPIED_ctr
```

The formatting of the information above the row of asterisks is subject to change, but hopefully provides a useful human-readable summary. Below the asterisks *all counters* maintained by the ticky-ticky system are dumped, in a format intended to be machine-readable: zero or more spaces, an integer, a space, the counter name, and a newline.

In fact, not *all* counters are necessarily dumped; compile- or run-time flags can render certain counters invalid. In this case, either the counter will simply not appear, or it will appear with a modified counter name, possibly along with an explanation for the omission (notice `ENT_PERM_IND_ctr` appears with an inserted `!` above). Software analysing this output should always check that it has the counters it expects. Also, beware: some of the counters can have *large* values!

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 C and crank up GCC: The native code-generator is designed to be quick, not mind-bogglingly clever. Better to let GCC have a go, as it tries much harder on register allocation, etc.

So, when we want very fast code, we use: `-O -fvia-C`.

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 7.13.8 and Section 7.13.9.

"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 4.6.7).

```
% 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 `__S <string>`. The `<string>` gives the strictness of the function's arguments. `L` is lazy (bad), `S` and `E` are strict (good), `P` is "primitive" (good), `U(...)` is strict and "unpackable" (very good), and `A` is absent (very good).

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 7.13.5.1.

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 4.9.2), 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 4.14.3).

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 are all enabled by options; by default GHC understands only plain Haskell 98.

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 flag control what variation of the language are permitted. Leaving out all of them gives you standard Haskell 98.

Generally speaking, all the language options are introduced by "`-X`", e.g. `-XTemplateHaskell`.

All the language options can be turned off by using the prefix "`No`"; e.g. "`-XNoTemplateHaskell`".

Language options recognised by Cabal can also be enabled using the `LANGUAGE` pragma, thus `{-# LANGUAGE TemplateHaskell #-}` (see Section 7.13.1>).

The flag `-fglasgow-exts` is equivalent to enabling the following extensions: `-XPrintExplicitForalls`, `-XForeignFunctionInterface`, `-XUnliftedFFITypes`, `-XGADTs`, `-XImplicitParams`, `-XScopedTypeVariables`, `-XUnboxedTuples`, `-XTypeSynonymInstances`, `-XStandaloneDeriving`, `-XDeriveDataTypeable`, `-XFlexibleContexts`, `-XFlexibleInstances`, `-XConstrainedClassMethods`, `-XMultiParamTypeClasses`, `-XFunctionalDependencies`, `-XMagicHash`, `-XPolymorphicComponents`, `-XExistentialQuantification`, `-XUnicodeSyntax`, `-XPostfixOperators`, `-XPatternGuards`, `-XLiberalTypeSynonyms`, `-XRankNTypes`, `-XImpredicativeTypes`, `-XTypeOperators`, `-XRecursiveDo`, `-XParallelListComp`, `-XEmptyDataDecls`, `-XKindSignatures`, `-XGeneralizedNewtypeDeriving`, `-XTypeFamilies`. 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 7.3.1).

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 7.3.1). 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 any such variable causes the entire pattern-match to become strict. For example:

```
data Foo = Foo Int Int#

f x = let (Foo a b, w) = ..rhs.. in ..body..
```

Since `b` has type `Int#`, the entire pattern match is strict, and the program behaves as if you had written

```
data Foo = Foo Int Int#

f x = case ..rhs.. of { (Foo a b, w) -> ..body.. }
```

7.2.2 Unboxed Tuples

Unboxed tuples aren't really exported by `GHC.Exts`, they're available by default with `-fglasgow-exts`. 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.

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 pretty stringent 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.
- No variable can have an unboxed tuple type, nor may a constructor or function argument have an unboxed tuple type. The following are all illegal:

```
data Foo = Foo (# Int, Int #)

f :: (# Int, Int #) -> (# Int, Int #)
f x = x

g :: (# Int, Int #) -> Int
g (# a,b #) = a

h x = let y = (# x,x #) in ...
```

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 o f{ (# 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 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 sematics 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.

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 98 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 98 integer lexeme followed by `##` is a `Word#`.
- `3.2#` has type `Float#`.
- `3.2##` has type `Double#`

7.3.2 New qualified operator syntax

A new syntax for referencing qualified operators is planned to be introduced by Haskell', and is enabled in GHC with the `-XNewQualifiedOperators` option. In the new syntax, the prefix form of a qualified operator is written *`module`*`.(`*`symbol`*`)` (in Haskell 98 this would be `(`*`module`*`.`*`symbol`*`)`), and the infix form is written `` ``*`module`*`.(`*`symbol`*`)` `` (in Haskell 98 this would be `` ``*`module`*`.`*`symbol`*`` ``). For example:

```
add x y = Prelude.(+) x y
subtract y = (`Prelude.(-)` y)
```

The new form of qualified operators is intended to regularise the syntax by eliminating odd cases like `Prelude..`. For example, when `NewQualifiedOperators` is on, it is possible to write the enumerated sequence `[Monday..]` without spaces, whereas in Haskell 98 this would be a reference to the operator '`.`' from module `Monday`.

When `-XNewQualifiedOperators` is on, the old Haskell 98 syntax for qualified operators is not accepted, so this option may cause existing Haskell 98 code to break.

7.3.3 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 4.6.3.

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

7.3.4 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.

7.3.5 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 :: Type -> TypeView

-- 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 7.3.4 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
```

    (We may lift this restriction in the future; the only cost is that type checking patterns would get a little more complicated.)

- 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.

7.3.6 The recursive do-notation

The recursive do-notation (also known as mdo-notation) is implemented as described in A recursive do for Haskell, by Levent Erkok, John Launchbury, Haskell Workshop 2002, pages: 29-37. Pittsburgh, Pennsylvania. This paper is essential reading for anyone making non-trivial use of mdo-notation, and we do not repeat it here.

The do-notation of Haskell 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 several applications can benefit from recursive bindings in the do-notation, and this extension provides the necessary syntactic support.

Here is a simple (yet contrived) example:

```
import Control.Monad.Fix

justOnes = mdo xs <- Just (1:xs)
               return xs
```

As you can guess `justOnes` will evaluate to `Just [1,1,1,...`.

The Control.Monad.Fix library introduces the `MonadFix` class. It's definition is:

```
class Monad m => MonadFix m where
   mfix :: (a -> m a) -> m a
```

The function `mfix` dictates how the required recursion operation should be performed. For example, `justOnes` desugars as follows:

```
justOnes = mfix (\xs' -> do { xs <- Just (1:xs'); return xs }
```

For full details of the way in which mdo is typechecked and desugared, see the paper A recursive do for Haskell. In particular, GHC implements the segmentation technique described in Section 3.2 of the paper.

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).

Here are some important points in using the recursive-do notation:

- The recursive version of the do-notation uses the keyword `mdo` (rather than `do`).
- It is enabled with the flag `-XRecursiveDo`, which is in turn implied by `-fglasgow-exts`.
- Unlike ordinary do-notation, but like `let` and `where` bindings, name shadowing is not allowed; that is, all the names bound in a single `mdo` must be distinct (Section 3.3 of the paper).
- Variables bound by a `let` statement in an `mdo` are monomorphic in the `mdo` (Section 3.1 of the paper). However GHC breaks the `mdo` into segments to enhance polymorphism, and improve termination (Section 3.2 of the paper).

Historical note: The old implementation of the mdo-notation (and most of the existing documents) used the name `MonadRec` for the class and the corresponding library. This name is not supported by GHC.

7.3.7 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 behavior 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.

7.3.8 Generalised (SQL-Like) List Comprehensions

Generalised list comprehensions are a further enhancement to the list comprehension syntatic sugar to allow operations such as sorting and grouping which are familiar from SQL. They are fully described in the paper Comprehensive comprehensions: comprehensions with "order by" and "group by", except that the syntax we use differs slightly from the paper.

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
, 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 function `sortWith` is not a keyword; it is an ordinary function that is 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 it's 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 by e
  ```

  This form of grouping is essentially the same as the one described above. However, since no function to use for the grouping has been supplied it will fall back on the `groupWith` function defined in GHC.Exts. This is the form of the group statement that we made use of in the opening example.

- ```
  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",...]
  ```

7.3.9 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 `-XNoImplicitPrelude` flag *also* 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.
- "Do" notation is translated using whatever functions `(>>=)`, `(>>)`, and `fail`, are in scope (not the Prelude versions). List comprehensions, mdo (Section 7.3.6), and parallel array comprehensions, are unaffected.
- Arrow notation (see Section 7.10) 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!

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.

7.3.10 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.

7.3.11 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.

7.3.12 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 puns and other patterns can be mixed in the same record:

```
data C = C {a :: Int, b :: Int}
f (C {a, b = 4}) = a
```

and that puns can be used wherever record patterns occur (e.g. in `let` bindings or at the top-level).

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}
```

Note that this expansion is purely syntactic, so the record pun expression refers to the nearest enclosing variable that is spelled the same as the field name.

7.3.13 Record wildcards

Record wildcards are enabled by the flag `-XRecordWildCards`.

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
```

Note that wildcards can be mixed with other patterns, including puns (Section 7.3.12); 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 expressions, writing, 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}
```

Note that this 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.

7.3.14 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 7.3.6), 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.

7.3.15 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.

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.

7.3.16 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: `-XScopedTypeVariables`, `-XLiberalTypeSynonyms`, `-XRank2Types`, `-XRankNTypes`, `-XPolymorphicComponents`, `-XExistentialQuantification`

`mdo` Stolen by: `-XRecursiveDo`,

`foreign` Stolen by: `-XForeignFunctionInterface`,

`rec`, `proc`, `-<`, `>-`, `-<<`, `>>-`, and `(|`, `|)` brackets Stolen by: `-XArrows`,

`?`*`varid`*, `%`*`varid` Stolen by: `-XImplicitParams`,

`[|`, `[e|`, `[p|`, `[d|`, `[t|`, `$(`, `$`*`varid` Stolen by: `-XTemplateHaskell`,

`[:`*`varid`*`|` Stolen by: `-XQuasiQuotes`,

***`varid`*`{#}`, *`char`*`#`, *`string`*`#`, *`integer`*`#`, *`float`*`#`, *`float`*`##`, `(#`, `#)`,** Stolen by: `-XMagicHash`,

7.4 Extensions to data types and type synonyms

7.4.1 Data types with no constructors

With the `-fglasgow-exts` flag, 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 $\star$ then an explicit kind annotation must be used (see Section 7.8.3).

Such data types have only one value, namely bottom. Nevertheless, they can be useful when defining "phantom types".

7.4.2 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
```

- A type variable can be an (unqualified) operator e.g. +. The lexical syntax is the same as that for variable operators, excluding "(.)", "(!)", and "(*)". In a binding position, the operator must be parenthesised. For example:

```
type T (+) = Int + Int
f :: T Either
f = Left 3

liftA2 :: Arrow (~>)
 => (a -> b -> c) -> (e ~> a) -> (e ~> b) -> (e ~> c)
liftA2 = ...
```

- 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.)

7.4.3 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.
- Unboxed tuple on left of an arrow.
- Partially-applied type synonym.

So, for example, this will be rejected:

```
type Pr = (# Int, Int #)

h :: Pr -> Int
h x = ...
```

because GHC does not allow unboxed tuples on the left of a function arrow.

7.4.4 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.

7.4.4.1 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.

7.4.4.2 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.

7.4.4.3 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 "##"
```

At the moment, record update syntax is only supported for Haskell 98 data types, so the following function does *not* work:

```
-- This is invalid; use explicit NewCounter instead for now
setTag :: Counter a -> a -> Counter a
setTag obj t = obj{ tag = t }
```

7.4.4.4 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!

7.4.5 Declaring data types with explicit constructor signatures

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 7.4.6, can only be declared using this form.

Notice that GADT-style syntax generalises existential types (Section 7.4.4). 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 7.4.4.2. 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 7.4.6).

- The type signature of each constructor is independent, and is implicitly universally quantified as usual. Different constructors may have different universally-quantified type variables and different type-class constraints. For example, this is fine:

```
data T a where
  T1 :: Eq b => b -> T b
  T2 :: (Show c, Ix c) => c -> [c] -> T c
```

- 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 Foo a :: (* -> *) -> * where ...
```

  The type variables (if given) may be explicitly kinded, so we could also write the header for `Foo` like this:

```
data Foo 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 )
```

- You can use record syntax on a GADT-style data type declaration:

```
data Person where
    Adult { name :: String, children :: [Person] } :: Person
    Child { name :: String } :: Person
```

  As usual, for every constructor that has a field `f`, the type of field `f` must be the same (modulo alpha conversion).

  At the moment, record updates are not yet possible with GADT-style declarations, so support is limited to record construction, selection and pattern matching. For example

```
aPerson = Adult { name = "Fred", children = [] }

shortName :: Person -> Bool
hasChildren (Adult { children = kids }) = not (null kids)
hasChildren (Child {})                  = False
```

- As in the case of existentials declared using the Haskell-98-like record syntax (Section 7.4.4.3), 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.

7.4.6 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 `-XRelaxedPolyRec`.

- A GADT can only be declared using GADT-style syntax (Section 7.4.5); 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's 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 7.4.5, 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 mechanism.

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. You must supply a context (in the example the context is `(Eq a)`), exactly as you would in an ordinary instance declaration. (In contrast the context is inferred in a `deriving` clause attached to a data type declaration.) 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.

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 7.6.3.1). 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`.

The stand-alone syntax is generalised for newtypes in exactly the same way that ordinary `deriving` clauses are generalised (Section 7.5.4). 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 clause for classes `Typeable` and `Data`

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 two more classes that may be automatically derived (provided the `-XDeriveDataTypeable` flag is specified): `Typeable`, and `Data`. These classes are defined in the library modules `Data.Typeable` and `Data.Generics` respectively, and the appropriate class must be in scope before it can be mentioned in the `deriving` clause.

An instance of `Typeable` can only be derived if the data type has seven or fewer type parameters, all of kind `*`. The reason for this is that the `Typeable` class is derived using the scheme described in Scrap More Boilerplate: Reflection, Zips, and Generalised Casts . (Section 7.4 of the paper describes the multiple `Typeable` classes that are used, and only `Typeable1` up to `Typeable7` are provided in the library.) In other cases, there is nothing to stop the programmer writing a `TypableX` class, whose kind suits that of the data type constructor, and then writing the data type instance by hand.

7.5.4 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!

7.5.4.1 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 `Parsers` 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.

7.5.4.2 A more precise specification

Derived instance declarations are constructed as follows. Consider the declaration (after expansion of any type synonyms)

```
newtype T v1...vn = T' (t vk+1...vn) deriving (c1...cm)
```

where

- The `ci` are partial applications of classes of the form `C t1'...tj'`, where the arity of `C` is exactly `j+1`. That is, `C` lacks exactly one type argument.
- The `k` is chosen so that `ci (T v1...vk)` is well-kinded.
- The type `t` is an arbitrary type.
- The type variables `vk+1...vn` do not occur in `t`, nor in the `ci`, and
- None of the `ci` is `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.

Then, for each `ci`, the derived instance declaration is:

```
instance ci t => ci (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.)

7.6 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 the design space (Simon Peyton Jones, Mark Jones, Erik Meijer).

All the extensions are enabled by the `-fglasgow-exts` flag.

7.6.1.1 Multi-parameter type classes

Multi-parameter type classes are permitted. 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

There are no restrictions on the context in a class declaration (which introduces superclasses), except 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`.)

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`).

7.6.2 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 7.8.1) 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

7.6.3.1 Relaxed rules for 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.

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 type constructor, and the `a1 ... an` are distinct type variables. Furthermore, 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 `-XFlexibleInstances` flag loosens these restrictions considerably. Firstly, multi-parameter type classes are permitted. Secondly, the context and head 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 7.6.3.2). 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 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
```

7.6.3.2 Undecidable instances

Sometimes even the rules of Section 7.6.3.1 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 7.6.3.1) 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*.

7.6.3.3 Overlapping instances

In general, *GHC requires that that it be unambiguous which instance declaration should be used to resolve a type-class constraint*. This behaviour can be modified by two flags: `-XOverlappingInstances` and `-XIncoherentInstances` , as this section discusses. Both these flags are dynamic flags, and can be set on a per-module basis, using an `OPTIONS_GHC` pragma if desired (Section 4.1.2).

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. For example, consider these declarations:

```
instance context1 => C Int a     where ...  -- (A)
instance context2 => C a   Bool  where ...  -- (B)
instance context3 => C Int [a]   where ...  -- (C)
instance context4 => C Int [Int] where ...  -- (D)
```

The instances (A) and (B) match the constraint `C Int Bool`, but (C) and (D) do not. When matching, GHC takes no account of the context of the instance declaration (`context1` etc). GHC's default behaviour is that *exactly one instance must match the constraint it is trying to resolve*. 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.

The `-XOverlappingInstances` flag instructs GHC to allow more than one instance to match, provided there is a most specific one. For example, the constraint `C Int [Int]` matches instances (A), (C) and (D), but the last is more specific, and hence is chosen. If there is no most-specific match, the program is rejected.

However, 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 Int [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 you add the flag `-XIncoherentInstances`, 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 Int [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.)

The willingness to be overlapped or incoherent is a property of the *instance declaration* itself, controlled by the presence or otherwise of the `-XOverlappingInstances` and `-XIncoherentInstances` flags when that module is being defined. Neither flag is required in a module that imports and uses the instance declaration. Specifically, during the lookup process:

- An instance declaration is ignored during the lookup process if (a) a more specific match is found, and (b) the instance declaration was compiled with `-XOverlappingInstances`. The flag setting for the more-specific instance does not matter.
- Suppose an instance declaration does not match the constraint being looked up, but does unify with it, so that it might match when the constraint is further instantiated. Usually GHC will regard this as a reason for not committing to some other constraint. But if the instance declaration was compiled with `-XIncoherentInstances`, GHC will skip the "does-it-unify?" check for that declaration.

These rules make it possible for a library author to design a library that relies on overlapping instances without the library client having to know.

If an instance declaration is compiled without `-XOverlappingInstances`, then that instance can never be overlapped. This could perhaps be inconvenient. Perhaps the rule should instead say that the *overlapping* instance declaration should be compiled in this way, rather than the *overlapped* one. Perhaps overlap at a usage site should be permitted regardless of how the instance declarations are compiled, if the `-XOverlappingInstances` flag is used at the usage site. (Mind you, the exact usage site can occasionally be hard to pin down.) We are interested to receive feedback on these points.

The `-XIncoherentInstances` flag implies the `-XOverlappingInstances` flag, but not vice versa.

7.6.3.4 Type synonyms in the instance head

Unlike Haskell 98, instance heads may use type synonyms. (The instance "head" is the bit after the "=>" in an instance decl.) As always, using a type synonym is just shorthand for writing the RHS of the type synonym definition. For example:

```
  type Point = (Int,Int)
  instance C Point   where ...
  instance C [Point] where ...
```

is legal. However, if you added

```
  instance C (Int,Int) where ...
```

as well, then the compiler will complain about the overlapping (actually, identical) instance declarations. As always, type synonyms must be fully applied. You cannot, for example, write:

```
  type P a = [[a]]
  instance Monad P where ...
```

This design decision is independent of all the others, and easily reversed, but it makes sense to me.

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 packed strings 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 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`.
- The standard defaulting rule (Haskell Report, Section 4.3.4) 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`.

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.

7.7 Type families

Indexed type families are a new GHC 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 two flavours: *data families* and *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 :: * -> *
```

7.7.1.1.1 Associated data family declarations

When a data family is declared as part of a type class, we drop the `family` special. The `GMap` declaration takes the following form

```
class GMapKey k where
  data GMap k :: * -> *
  ...
```

In contrast to toplevel declarations, named arguments must be used for all type parameters that are to be used as type-indexes. Moreover, the argument names must be 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
  data T c a :: *
```

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 legit 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. This implies that the number of parameters of an instance declaration matches the arity determined by the kind of the family. Although, all data families are declared with the `data` keyword, instances can be either `data` or `newtype`s, or a mix of both.

Even if type families are defined as toplevel declarations, functions that perform different computations for different family instances 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
nonsence :: T a -> Int
nonsence A = 1              -- WRONG: These two equations together...
nonsence B = 2              -- ...will produce a type error.
```

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.

7.7.1.2.1 Associated data instances

When an associated data family instance is declared within a type class instance, we drop the `instance` keyword in the family instance. So, the `Either` instance for `GMap` becomes:

```
instance (GMapKey a, GMapKey b) => GMapKey (Either a b) where
  data GMap (Either a b) v = GMapEither (GMap a v) (GMap b v)
  ...
```

The most important point about associated family instances is that the type indexes corresponding to class parameters must be identical to the type given in the instance head; here this is the first argument of `GMap`, namely `Either a b`, which coincides with the only class parameter. Any parameters to the family constructor that do not correspond to class parameters, need to be variables in every instance; here this is the variable `v`.

Instances for an associated family can only appear as part of instances declarations of the class in which the family was declared - just as with the equations of the methods of a class. Also in correspondence to how methods are handled, declarations of associated types can be omitted in class instances. If an associated family instance is omitted, the corresponding instance type is not inhabited; i.e., only diverging expressions, such as `undefined`, can assume the type.

7.7.1.2.2 Scoping of class parameters

In the case of multi-parameter type classes, the visibility of class parameters in the right-hand side of associated family instances depends *solely* on the parameters of the data 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.

7.7.1.2.3 Type class instances of family instances

Type class instances of instances of data families can be defined as usual, and in particular data instance declarations can 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 ...
```

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.

7.7.1.2.4 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.

7.7.1.3 Import and export

The association of data constructors with type families is more dynamic than that is the case with standard data and newtype declarations. In the standard case, the notation `T(..)` in an import or export list denotes the type constructor and all the data constructors introduced in its declaration. However, a family declaration never introduces any data constructors; instead, data constructors are introduced by family instances. As a result, which data constructors are associated with a type family depends on the currently visible instance declarations for that family. Consequently, an import or export item of the form `T(..)` denotes the family constructor and all currently visible data constructors - in the case of an export item, these may be either imported or defined in the current module. The treatment of import and export items that explicitly list data constructors, such as `GMap(GMapEither)`, is analogous.

7.7.1.3.1 Associated families

As expected, an import or export item of the form `C(..)` denotes all of the class' methods and associated types. However, when associated types are explicitly listed as subitems of a class, we need some new syntax, as uppercase identifiers as subitems are usually data constructors, not type constructors. To clarify that we denote types here, each associated type name needs to be prefixed by the keyword `type`. So for example, when explicitly listing the components of the `GMapKey` class, we write `GMapKey(type GMap, empty, lookup, insert)`.

7.7.1.3.2 Examples

Assuming our running `GMapKey` class example, let us look at some export lists and their meaning:

- `module GMap (GMapKey) where...`: Exports just the class name.
- `module GMap (GMapKey(..)) where...`: Exports the class, the associated type `GMap` and the member functions `empty`, `lookup`, and `insert`. None of the data constructors is exported.
- `module GMap (GMapKey(..), GMap(..)) where...`: As before, but also exports all the data constructors `GMapInt`, `GMapChar`, `GMapUnit`, `GMapPair`, and `GMapUnit`.
- `module GMap (GMapKey(empty, lookup, insert), GMap(..)) where...`: As before.
- `module GMap (GMapKey, empty, lookup, insert, GMap(..)) where...`: As before.

Finally, you can write `GMapKey(type GMap)` to denote both the class `GMapKey` as well as its associated type `GMap`. However, 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.

7.7.1.3.3 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.

7.7.2 Synonym families

Type 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 type synonyms). 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.

7.7.2.1 Type family declarations

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 it's 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
```

7.7.2.1.1 Associated type family declarations

When a type family is declared as part of a type class, we drop the `family` special. The `Elem` declaration takes the following form

```
class Collects ce where
  type Elem ce :: *
  ...
```

The argument names of the type family must be 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 :: *
```

These rules are exactly as for associated data families.

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. 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 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
```

7.7.2.2.1 Associated type instance declarations

When an associated family instance is declared within a type class instance, we drop the `instance` keyword in the family instance. So, the `[e]` instance for `Elem` becomes:

```
instance (Eq (Elem [e])) => Collects ([e]) where
  type Elem [e] = e
  ...
```

The most important point about associated family instances is that the type indexes corresponding to class parameters must be identical to the type given in the instance head; here this is `[e]`, which coincides with the only class parameter.

Instances for an associated family can only appear as part of instances declarations of the class in which the family was declared - just as with the equations of the methods of a class. Also in correspondence to how methods are handled, declarations of associated types can be omitted in class instances. If an associated family instance is omitted, the corresponding instance type is not inhabited; i.e., only diverging expressions, such as `undefined`, can assume the type.

7.7.2.2.2 Overlap of type synonym instances

The instance declarations of a type family used in a single program may only overlap if the right-hand sides of the overlapping instances coincide for the overlapping types. More formally, two instance declarations overlap if there is a substitution that makes the left-hand sides of the instances syntactically the same. Whenever that is the case, the right-hand sides of the instances must also be syntactically equal under the same substitution. This 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.

Here are two example to illustrate the condition under which overlap is permitted.

```
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]
```

7.7.2.2.3 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 Functions"). Instance declarations have the general form

```
type instance F t1 .. tn = t
```

where we require that for every type family application `(G s1 .. sm)` in `t`,

1. `s1 .. sm` do not contain any type family constructors,
2. the total number of symbols (data type constructors and type variables) in `s1 .. sm` is strictly smaller than in `t1 .. tn`, and
3. for every type variable `a`, `a` occurs in `s1 .. sm` at most as often as in `t1 .. tn`.

These restrictions are easily verified and ensure termination of type inference. However, they are not sufficient to guarantee completeness of type inference in the presence of, so called, "loopy equalities", such as `a ~ [F a]`, where a recursive occurrence of a type variable is underneath a family application and data constructor application - see the above mentioned paper for details.

If the option `-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.

7.7.2.3 Equality constraints

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.

NB: Equalities in superclass contexts are not fully implemented in GHC 6.10.

7.8 Other type system extensions

7.8.1 Type signatures

7.8.1.1 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 type-variable ...))*. With `-XFlexibleContexts` these type signatures are perfectly OK

```
g :: Eq [a] => ...
g :: Ord (T a ()) => ...
```

GHC imposes the following restrictions on the constraints in a type signature. Consider the type:

```
forall tv1..tvn (c1, ...,cn) => type
```

(Here, we write the "foralls" explicitly, although the Haskell source language omits them; in Haskell 98, all the free type variables of an explicit source-language type signature are universally quantified, except for the class type variables in a class declaration. However, in GHC, you can give the foralls if you want. See Section 7.8.4).

1. *Each universally quantified type variable* `tvi` *must be reachable from* `type`. A type variable `a` is "reachable" if it appears in the same constraint as either a type variable free in `type`, or another reachable type variable. A value with a type that does not obey this reachability restriction cannot be used without introducing ambiguity; that is why the type is rejected. Here, for example, is an illegal type:

   ```
   forall a. Eq a => Int
   ```

   When a value with this type was used, the constraint `Eq tv` would be introduced where `tv` is a fresh type variable, and (in the dictionary-translation implementation) the value would be applied to a dictionary for `Eq tv`. The difficulty is that we can never know which instance of `Eq` to use because we never get any more information about `tv`.

   Note that the reachability condition is weaker than saying that `a` is functionally dependent on a type variable free in `type` (see Section 7.6.2). The reason for this is there might be a "hidden" dependency, in a superclass perhaps. So "reachable" is a conservative approximation to "functionally dependent". For example, consider:

   ```
   class C a b | a -> b where ...
   class C a b => D a b where ...
   f :: forall a b. D a b => a -> a
   ```

   This is fine, because in fact `a` does functionally determine `b` but that is not immediately apparent from `f`'s type.

2. *Every constraint* `ci` *must mention at least one of the universally quantified type variables* `tvi`. For example, this type is OK because `C a b` mentions the universally quantified type variable `b`:

   ```
   forall a. C a b => burble
   ```

   The next type is illegal because the constraint `Eq b` does not mention `a`:

   ```
   forall a. Eq b => burble
   ```

   The reason for this restriction is milder than the other one. The excluded types are never useful or necessary (because the offending context doesn't need to be witnessed at this point; it can be floated out). Furthermore, floating them out increases sharing. Lastly, excluding them is a conservative choice; it leaves a patch of territory free in case we need it later.

7.8.2 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
```

7.8.2.1 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`.

7.8.2.2 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
  ```

7.8.2.3 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.

7.8.2.4 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`.

7.8.3 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.

7.8.4 Arbitrary-rank polymorphism

Haskell type signatures are implicitly quantified. The new 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.

However, 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.

GHC has three flags to control higher-rank types:

- `-XPolymorphicComponents`: data constructors (only) can have polymorphic argument types.
- `-XRank2Types`: any function (including data constructors) can have a rank-2 type.
- `-XRankNTypes`: any function (including data constructors) can have an arbitrary-rank type. That is, you can nest `forall`s arbitrarily deep in function arrows. In particular, a forall-type (also called a "type scheme"), including an operational 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 7.8.6)

Of course `forall` becomes a keyword; you can't use `forall` as a type variable any more!

7.8.4.1 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 (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 :: (Ord a => [a] -> [a]) -> Swizzle
```

Notice that you don't need to use a `forall` if there's an explicit context. For example in the first argument of the constructor `MkSwizzle`, an implicit "`forall a.`" is prefixed to the argument type. The implicit `forall` quantifies all type variables that are not already in scope, and are mentioned in the type quantified over.

As for type signatures, implicit quantification happens for non-overloaded types too. So if you write this:

```
  data T a = MkT (Either a b) (b -> b)
```

it's just as if you had written this:

```
  data T a = MkT (forall b. Either a b) (forall b. b -> b)
```

That is, since the type variable `b` isn't in scope, it's implicitly universally quantified. (Arguably, it would be better to *require* explicit quantification on constructor arguments where that is what is wanted. Feedback welcomed.)

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.

7.8.4.2 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 7.8.6), 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.

7.8.4.3 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.

7.8.5 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.

7.8.6 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; it scopes over the entire definition of `f`. In particular, it is in scope at 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`.

Note: GHC 6.6 contains substantial changes to the way that scoped type variables work, compared to earlier releases. Read this section carefully!

7.8.6.1 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 7.8.6.2)

- An expression type signature (Section 7.8.6.3)
- A pattern type signature (Section 7.8.6.4)
- Class and instance declarations (Section 7.8.6.5)

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)
```

7.8.6.2 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 "`f`", 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.

7.8.6.3 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 a.  ST s Bool` brings the type variable `s` into scope, in the annotated expression `(op >>= \(x ::  STRef s Int) -> g x)`.

7.8.6.4 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.

7.8.6.5 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
```

7.8.7 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
```

7.9 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 7.9.3) 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.

7.9.1 Syntax

Template Haskell has the following new syntactic constructions. You need to use the flag `-XTemplateHaskell` to switch these syntactic extensions on (`-XTemplateHaskell` is no longer implied by `-fglasgow-exts`).

- 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 list of top-level declarations; the spliced expression must have type `Q [Dec]`

  Inside a splice you can can only call functions defined in imported modules, not functions defined elsewhere in the same module.

- A expression quotation is written in Oxford brackets, thus:
  - `[| ... |]`, 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 Typ`.
- 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 7.9.5.
- 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.
  - `''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.

(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]`. Type splices are not implemented, and neither are pattern splices or quotations.

7.9.2 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.

7.9.3 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
```

7.9.4 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.

7.9.5 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.

In the example, the quasiquoter `expr` is bound to a value of type `Language.Haskell.TH.Quote.QuasiQuoter` which contains two functions for quoting expressions and patterns, respectively. The first argument to each quoter is the (arbitrary) string enclosed in the Oxford brackets. The context of the quasi-quotation statement determines which of the two parsers is called: if the quasi-quotation occurs in an expression context, the expression parser is called, and if it occurs in a pattern context, the pattern parser is called.

Note that in the example we make 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`.

In general, a quasi-quote has the form `[$`*`quoter`*`| `*`string`*` |]`. The *`quoter`* must be the name of an imported quoter; it cannot be an arbitrary expression. The quoted *`string`* can be arbitrary, and may contain newlines.

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.

```
{- 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 ()
              }
          }


{- 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 parseExprExp 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
```

7.10 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.

7.10.1 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 7.14) 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.

7.10.2 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) >>> f) ||| (arr (\y -> y+2) >>> g)
```

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.

7.10.3 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 `ArrowChoice` class includes a combinator

```
ArrowChoice a => (<+>) :: a e c -> a e c -> a e 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)
```

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 k >>> (f <+> g) = (arr k >>> f) <+> (arr 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 operator must also not use any variable defined within the current arrow abstraction.

We could define our own operator

```
untilA :: ArrowChoice a => a e () -> a e Bool -> a e ()
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)|)
```

7.10.4 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 c -> a (e,Ex) c -> a e 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, this stack was empty. In the second argument to `handleA`, this stack consists of one value, the value of the exception. The command form of lambda merely gives this value a name.

More concretely, the values on the stack are paired to the right of the environment. So operators like `handleA` that pass extra inputs to their subcommands can be designed for use with the notation by pairing the values 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 b -> a (e,b) c -> a (e,c) d -> a e d
runReader :: ... => a e c -> a' (e,State) c
runState :: ... => a e c -> a' (e,State) (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 b -> a (e,b) c -> a e c
u `bind` f = returnA &&& u >>> f

bind_ :: Arrow a => a e b -> a e c -> a e c
u `bind_` f = u `bind` (arr fst >>> f)
```

We could simulate `if` by defining

```
cond :: ArrowChoice a => a e b -> a e b -> a (e,Bool) b
cond f g = arr (\ (e,b) -> if b then Left e else Right e) >>> f ||| g
```

7.10.5 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.

7.10.6 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.

7.11 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`.

7.11.1 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. For example:

```
let ![x,y] = e in b
```

is a strict binding: operationally, it evaluates `e`, matches it against the pattern `[x,y]`, and then evaluates `b`. (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*.) 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`.

7.11.2 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 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.

7.12 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.

Assertion failures can be caught, see the documentation for the `Control.Exception` library for the details.

7.13 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.

7.13.1 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 7.13).

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-languages` (see Section 4.4).

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.

7.13.2 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 4.1.2 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 7.13).

7.13.3 INCLUDE pragma

The `INCLUDE` pragma is for specifying the names of C header files that should be `#include`'d into the C source code generated by the compiler for the current module (if compiling via C). For example:

```
{-# INCLUDE "foo.h" #-}
{-# INCLUDE <stdio.h> #-}
```

`INCLUDE` is a file-header pragma (see Section 7.13).

An `INCLUDE` pragma is the preferred alternative to the `-#include` option (Section 4.10.5), because the `INCLUDE` pragma is understood by other compilers. Yet another alternative is to add the include file to each `foreign import` declaration in your code, but we don't recommend using this approach with GHC.

7.13.4 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 7.4.2).

Warnings and deprecations are not reported for (a) uses within the defining module, and (b) 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`.

7.13.5 INLINE and NOINLINE pragmas

These pragmas control the inlining of function definitions.

7.13.5.1 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. Normally, if GHC decides a function is "too expensive" to inline, it will not do so, nor will it export that unfolding for other modules to use.

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:

- No functions are inlined into `f`. Otherwise GHC might inline a big function into `f`'s right hand side, making `f` big; and then inline `f` blindly.
- The float-in, float-out, and common-sub-expression transformations are not applied to the body of `f`.
- An INLINE function is not worker/wrappered by strictness analysis. It's going to be inlined wholesale instead.

All of these effects are aimed at ensuring that what gets inlined is exactly what you asked for, no more and no less.

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` pragma (Section 7.13.5.2).

Note: the HBC compiler doesn't like `INLINE` pragmas, so if you want your code to be HBC-compatible you'll have to surround the pragma with C pre-processor directives `#ifdef __GLASGOW_HASKELL__...#endif`.

7.13.5.2 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).

7.13.5.3 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 7.14).

7.13.6 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.

7.13.7 RULES pragma

The RULES pragma lets you specify rewrite rules. It is described in Section 7.14.

7.13.8 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.

A `SPECIALIZE` has the effect of generating (a) a specialised version of the function and (b) a rewrite rule (see Section 7.14) that rewrites a call to the un-specialised function into a call to the specialised one.

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.

A `SPECIALIZE` pragma can optionally be followed with a `INLINE` or `NOINLINE` pragma, optionally followed by a phase, as described in Section 7.13.5. 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.

Warning: you can make GHC diverge by using `SPECIALISE INLINE` on an ordinarily-recursive function.

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 7.14.4).

7.13.9 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.

Compatible with HBC, by the way, except perhaps in the placement of the pragma.

7.13.10 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`, 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.

If a field cannot be unpacked, you will not get a warning, so it might be an idea to check the generated code with `-ddump-simpl`.

See also the `-funbox-strict-fields` flag, which essentially has the effect of adding `{-# UNPACK #-}` to every strict constructor field.

7.13.11 SOURCE pragma

The `{-# SOURCE #-}` pragma is used only in `import` declarations, to break a module loop. It is described in detail in Section 4.6.9.

7.14 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 in detail.

7.14.1 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 7.13.5.3), 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.
- 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 7.8.6.
- 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 7.14.2).

7.14.2 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 4.9.2.) 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 7.14.4 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.

- Ordinary inlining happens at the same time as rule rewriting, which may lead to unexpected results. Consider this (artificial) example

```
f x = x
{-# RULES "f" f True = False #-}

g y = f y

h z = g True
```

  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 on `f`, to ensure that it is not inlined until its RULEs have had a chance to fire.

7.14.3 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` 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`
- `sortBy`

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.

7.14.4 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.

7.14.5 Controlling what's going on

- Use `-ddump-rules` to see what transformation rules GHC is using.
- Use `-ddump-simpl-stats` to see what rules are being fired. If you add `-dppr-debug` you get a 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`.

7.14.6 CORE pragma

The external core format supports 'Note' annotations; the `CORE` pragma gives a way to specify what these should be in your Haskell source code. Syntactically, core annotations are attached to expressions and take a Haskell string literal as an argument. The following function definition shows an example:

```
f x = ({-# CORE "foo" #-} show) ({-# CORE "bar" #-} x)
```

Semantically, this is equivalent to:

```
g x = show x
```

However, when external core is generated (via `-fext-core`), there will be Notes attached to the expressions `show` and `x`. The core function declaration for `f` is:

```
  f :: %forall a . GHCziShow.ZCTShow a ->
                   a -> GHCziBase.ZMZN GHCziBase.Char =
    \ @ a (zddShow::GHCziShow.ZCTShow a) (eta::a) ->
        (%note "foo"
         %case zddShow %of (tpl::GHCziShow.ZCTShow a)
           {GHCziShow.ZCDShow
            (tpl1::GHCziBase.Int ->
                   a ->
                   GHCziBase.ZMZN GHCziBase.Char -> GHCziBase.ZMZN GHCziBase.Cha
r)
            (tpl2::a -> GHCziBase.ZMZN GHCziBase.Char)
            (tpl3::GHCziBase.ZMZN a ->
                   GHCziBase.ZMZN GHCziBase.Char -> GHCziBase.ZMZN GHCziBase.Cha
r) ->
              tpl2})
        (%note "bar"
         eta);
```

Here, we can see that the function `show` (which has been expanded out to a case expression over the Show dictionary) has a `%note` attached to it, as does the expression `eta` (which used to be called `x`).

7.15 Special built-in functions

GHC has a few built-in functions with special behaviour. These are now described in the module GHC.Prim in the library documentation.

7.16 Generic classes

The ideas behind this extension are described in detail in "Derivable type classes", Ralf Hinze and Simon Peyton Jones, Haskell Workshop, Montreal Sept 2000, pp94-105. An example will give the idea:

```
  import Generics

  class Bin a where
    toBin   :: a -> [Int]
    fromBin :: [Int] -> (a, [Int])

    toBin {| Unit |}    Unit    = []
    toBin {| a :+: b |} (Inl x)   = 0 : toBin x
    toBin {| a :+: b |} (Inr y)   = 1 : toBin y
    toBin {| a :*: b |} (x :*: y) = toBin x ++ toBin y

    fromBin {| Unit |}    bs      = (Unit, bs)
    fromBin {| a :+: b |} (0:bs)  = (Inl x, bs')    where (x,bs') = fromBin bs
    fromBin {| a :+: b |} (1:bs)  = (Inr y, bs')    where (y,bs') = fromBin bs
    fromBin {| a :*: b |} bs    = (x :*: y, bs'') where (x,bs' ) = fromBin bs
                (y,bs'') = fromBin bs'
```

This class declaration explains how `toBin` and `fromBin` work for arbitrary data types. They do so by giving cases for unit, product, and sum, which are defined thus in the library module `Generics`:

```
data Unit    = Unit
data a :+: b = Inl a | Inr b
data a :*: b = a :*: b
```

Now you can make a data type into an instance of Bin like this:

```
instance (Bin a, Bin b) => Bin (a,b)
instance Bin a => Bin [a]
```

That is, just leave off the "where" clause. Of course, you can put in the where clause and over-ride whichever methods you please.

7.16.1 Using generics

To use generics you need to

- Use the flags `-fglasgow-exts` (to enable the extra syntax), `-XGenerics` (to generate extra per-data-type code), and `-package lang` (to make the `Generics` library available.
- Import the module `Generics` from the `lang` package. This import brings into scope the data types `Unit`, `:*:`, and `:+:`. (You don't need this import if you don't mention these types explicitly; for example, if you are simply giving instance declarations.)

7.16.2 Changes wrt the paper

Note that the type constructors `:+:` and `:*:` can be written infix (indeed, you can now use any operator starting in a colon as an infix type constructor). Also note that the type constructors are not exactly as in the paper (Unit instead of 1, etc). Finally, note that the syntax of the type patterns in the class declaration uses "`{|`" and "`|}`" brackets; curly braces alone would ambiguous when they appear on right hand sides (an extension we anticipate wanting).

7.16.3 Terminology and restrictions

Terminology. A "generic default method" in a class declaration is one that is defined using type patterns as above. A "polymorphic default method" is a default method defined as in Haskell 98. A "generic class declaration" is a class declaration with at least one generic default method.

Restrictions:

- Alas, we do not yet implement the stuff about constructor names and field labels.
- A generic class can have only one parameter; you can't have a generic multi-parameter class.
- A default method must be defined entirely using type patterns, or entirely without. So this is illegal:

```
class Foo a where
  op :: a -> (a, Bool)
  op {| Unit |} Unit = (Unit, True)
  op x               = (x,    False)
```

  However it is perfectly OK for some methods of a generic class to have generic default methods and others to have polymorphic default methods.

- The type variable(s) in the type pattern for a generic method declaration scope over the right hand side. So this is legal (note the use of the type variable "p" in a type signature on the right hand side:

```
class Foo a where
  op :: a -> Bool
  op {| p :*: q |} (x :*: y) = op (x :: p)
  ...
```

- The type patterns in a generic default method must take one of the forms:

```
      a :+: b
      a :*: b
      Unit
```

  where "a" and "b" are type variables. Furthermore, all the type patterns for a single type constructor (`:*:`, say) must be identical; they must use the same type variables. So this is illegal:

```
class Foo a where
  op :: a -> Bool
  op {| a :+: b |} (Inl x) = True
  op {| p :+: q |} (Inr y) = False
```

  The type patterns must be identical, even in equations for different methods of the class. So this too is illegal:

```
class Foo a where
  op1 :: a -> Bool
  op1 {| a :*: b |} (x :*: y) = True

  op2 :: a -> Bool
  op2 {| p :*: q |} (x :*: y) = False
```

  (The reason for this restriction is that we gather all the equations for a particular type constructor into a single generic instance declaration.)

- A generic method declaration must give a case for each of the three type constructors.
- The type for a generic method can be built only from:
  - Function arrows
  - Type variables
  - Tuples
  - Arbitrary types not involving type variables

  Here are some example type signatures for generic methods:

```
  op1 :: a -> Bool
  op2 :: Bool -> (a,Bool)
  op3 :: [Int] -> a -> a
  op4 :: [a] -> Bool
```

  Here, op1, op2, op3 are OK, but op4 is rejected, because it has a type variable inside a list.

  This restriction is an implementation restriction: we just haven't got around to implementing the necessary bidirectional maps over arbitrary type constructors. It would be relatively easy to add specific type constructors, such as Maybe and list, to the ones that are allowed.

- In an instance declaration for a generic class, the idea is that the compiler will fill in the methods for you, based on the generic templates. However it can only do so if
  - The instance type is simple (a type constructor applied to type variables, as in Haskell 98).
  - No constructor of the instance type has unboxed fields.

  (Of course, these things can only arise if you are already using GHC extensions.) However, you can still give an instance declarations for types which break these rules, provided you give explicit code to override any generic default methods.

The option `-ddump-deriv` dumps incomprehensible stuff giving details of what the compiler does with generic declarations.

7.16.4 Another example

Just to finish with, here's another example I rather like:

```
class Tag a where
  nCons :: a -> Int
  nCons {| Unit |}    _ = 1
  nCons {| a :*: b |} _ = 1
  nCons {| a :+: b |} _ = nCons (bot::a) + nCons (bot::b)

  tag :: a -> Int
  tag {| Unit |}    _       = 1
  tag {| a :*: b |} _       = 1
  tag {| a :+: b |} (Inl x) = tag x
  tag {| a :+: b |} (Inr y) = nCons (bot::a) + tag y
```

7.17 Control over monomorphism

GHC supports two flags that control the way in which generalisation is carried out at let and where bindings.

7.17.1 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`.

7.17.2 Monomorphic pattern bindings

As an experimental change, we are exploring the possibility of making pattern bindings monomorphic; that is, not generalised at all. A pattern binding is a binding whose LHS has no function arguments, and is not a simple variable. For example:

```
f x = x                    -- Not a pattern binding
f = \x -> x                -- Not a pattern binding
f :: Int -> Int = \x -> x  -- Not a pattern binding

(g,h) = e                  -- A pattern binding
(f) = e                    -- A pattern binding
[x] = e                    -- A pattern binding
```

Experimentally, GHC now makes pattern bindings monomorphic *by default*. Use `-XNoMonoPatBinds` to recover the standard behaviour.

7.18 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.

7.18.1 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.

7.18.2 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 STM is Control.Concurrent.STM. 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.

7.18.3 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 4.12). 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).

7.18.4 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 7.18.1), 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 Control.Parallel:

```
infixr 0 `par`
infixr 1 `seq`

par :: a -> b -> b
seq :: 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 (seq 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 `seq` 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 `seq` 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.

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.

More sophisticated combinators for expressing parallelism are available from the Control.Parallel.Strategies module. This module builds functionality around `par`, expressing more elaborate patterns of parallel computation, such as parallel `map`.

7.18.5 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.

Chapter 8

Foreign function interface (FFI)

GHC (mostly) conforms to the Haskell 98 Foreign Function Interface Addendum 1.0, whose definition is available from http://www.haske
.

To enable FFI support in GHC, give 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
```

8.2 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 4.6.4.

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

#ifdef __GLASGOW_HASKELL__
extern void __stginit_Foo ( void );
#endif

int main(int argc, char *argv[])
{
  int i;
```

```
  hs_init(&argc, &argv);
#ifdef __GLASGOW_HASKELL__
  hs_add_root(__stginit_Foo);
#endif

  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`).

Next, we call `hs_add_root`, a GHC-specific interface which is required to initialise the Haskell modules in the program. The argument to `hs_add_root` should be the name of the initialization function for the "root" module in your program - in other words, the module which directly or indirectly imports all the other Haskell modules in the program. In a standalone Haskell program the root module is normally `Main`, but when you are using Haskell code from a library it may not be. If your program has multiple root modules, then you can call `hs_add_root` multiple times, one for each root. The name of the initialization function for module *M* is `__stginit_`*M*, and it may be declared as an external function symbol as in the code above. Note that the symbol name should be transformed according to the Z-encoding:

Character	Replacement
.	zd
_	zu
`	zq
Z	ZZ
z	zz

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.

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:

```
HsBool mylib_init(void){
  int argc = ...
  char *argv[] = ...

  // Initialize Haskell runtime
```

[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 12.1.3.

```
  hs_init(&argc, &argv);

  // Tell Haskell about all root modules
  hs_add_root(__stginit_Foo);

  // 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()` and `hs_add_root()` as normal to initialise the Haskell runtime, and the corresponding deinitialisation function `mylib_end()` calls `hs_exit()` to shut down the runtime.

8.2.1.3 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).

8.2.2 Using function headers

C functions are normally declared using prototypes in a C header file. Earlier versions of GHC (6.8.3 and earlier) `#included` the header file in the C source file generated from the Haskell code, and the C compiler could therefore check that the C function being called via the FFI was being called at the right type.

GHC no longer includes external header files when compiling via C, so this checking is not performed. The change was made for compatibility with the native code backend (`-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.

Chapter 9

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 12.1, which describes Glasgow Haskell's shortcomings vs. the Haskell language definition, may also be of interest.)

9.1 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.

9.2 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 `-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 4.6.11.

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 -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 10

Other Haskell utility programs

This section describes other program(s) which we distribute, that help with the Great Haskell Programming Task.

10.1 Ctags and Etags for Haskell: hasktags

hasktags is a very simple Haskell program that produces ctags "tags" and etags "TAGS" files for Haskell programs.

When loaded into an editor such an NEdit, Vim, or Emacs, this allows one to easily navigate around a multi-file program, finding definitions of functions, types, and constructors.

Invocation Syntax:

```
hasktags files
```

This will read all the files listed in `files` and produce a ctags "tags" file and an etags "TAGS" file in the current directory.

Example usage

```
find -name \*.\*hs | xargs hasktags
```

This will find all haskell source files in the current directory and below, and create tags files indexing them in the current directory.

hasktags is a simple program that uses simple parsing rules to find definitions of functions, constructors, and types. It isn't guaranteed to find everything, and will sometimes create false index entries, but it usually gets the job done fairly well. In particular, at present, functions are only indexed if a type signature is given for them.

Before hasktags, there used to be **fptags** and **hstags**, which did essentially the same job, however neither of these seem to be maintained any more.

10.1.1 Using tags with your editor

With NEdit, load the "tags" file using "File/Load Tags File". Use "Ctrl-D" to search for a tag.

With XEmacs, load the "TAGS" file using "visit-tags-table". Use "M-." to search for a tag.

10.2 "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.

10.3 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.

10.3.1 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: **ghc**)

`-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`.

`-?` 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.

10.3.2 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`.

Note
A `foreign import` of a C function may be inlined across a module boundary, in which case you must arrange for the importing module to `#include` the C header file generated by **hsc2hs** (see Section 8.2.2). For this reason we avoid using `#def` in the libraries.

`#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.

10.3.3 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.

Chapter 11

Running GHC on Win32 systems

11.1 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".

11.2 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.

11.3 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!

11.4 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.

11.5 Using GHC (and other GHC-compiled executables) with cygwin

11.5.1 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`).

11.5.2 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.

11.5.3 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
```

11.6 Building and using Win32 DLLs

Making Haskell libraries into DLLs doesn't work on Windows at the moment; we hope to re-instate this facility in the future. Note that building an entire Haskell application as a single DLL is still supported: it's just multi-DLL Haskell programs that don't work. The Windows distribution of GHC contains static libraries only.

11.6.1 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.

11.6.2 Making DLLs to be called from other languages

If you want to package up Haskell code to be called from other languages, such as Visual Basic or C++, there are some extra things it is useful to know. 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,

```
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
```

- Compile it up:

```
ghc -c adder.hs -fglasgow-exts
```

  This will produce two files, adder.o and adder_stub.o

- compile up a `DllMain()` that starts up the Haskell RTS-—a possible implementation is:

```
#include <windows.h>
#include <Rts.h>

extern void __stginit_Adder(void);

static char* args[] = { "ghcDll", NULL };
                       /* N.B. argv arrays must end with NULL */
BOOL
STDCALL
DllMain
   ( HANDLE hModule
   , DWORD reason
   , void* reserved
   )
{
  if (reason == DLL_PROCESS_ATTACH) {
      /* By now, the RTS DLL should have been hoisted in, but we need to start it up. */
      startupHaskell(1, args, __stginit_Adder);
      return TRUE;
  }
  return TRUE;
}
```

  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 this up:

```
ghc -c dllMain.c
```

- Construct the DLL:

```
ghc -shared -o adder.dll adder.o adder_stub.o dllMain.o
```

- Start using `adder` from VBA-—here's how I would `Declare` it:

```
Private Declare Function adder Lib "adder.dll" Alias "adder@8"
      (ByVal x As Long, ByVal y As Long) As Long
```

  Since this Haskell DLL depends on a couple of the DLLs that come with GHC, make sure that they are in scope/visible.

  Building statically linked DLLs is the same as in the previous section: it suffices to add `-static` to the commands used to compile up the Haskell source and build the DLL.

11.6.3 Beware of DllMain()!

The body of a `DllMain()` function is an extremely dangerous place! This is because the order in which DLLs are unloaded when a process is terminating is unspecified. This means that the `DllMain()` for your DLL may be called when other DLLs containing functions that you call when de-initializing your DLL have already been unloaded. In other words, you can't put shutdown code inside `DllMain()`, unless your shutdown code only requires use of certain functions which are guaranteed to be available (see the Platform SDK docs for more info).

In particular, if you are writing a DLL that's statically linked with Haskell, it is not safe to call `hs_exit()` from `DllMain()`, since `hs_exit()` may make use of other DLLs (see also Section 8.2.1.3). What's more, if you wait until program shutdown to execute your deinitialisation code, Windows will have terminated all the threads in your program except the one calling `DllMain()`, which can cause even more problems.

A solution is to always export `Begin()` and `End()` functions from your DLL, and call these from the application that uses the DLL, so that you can be sure that all DLLs needed by any shutdown code in your End() function are available when it is called.

The following example is untested but illustrates the idea (please let us know if you find problems with this example or have a better one). Suppose we have a DLL called Lewis which makes use of 2 Haskell modules `Bar` and `Zap`, where `Bar` imports `Zap` and is therefore the root module in the sense of Section 8.2.1.1. Then the main C++ unit for the DLL would look something like:

```
// Lewis.cpp -- compiled using GCC
#include <Windows.h>
#include "HsFFI.h"

#define __LEWIS_DLL_EXPORT
#include "Lewis.h"

#include "Bar_stub.h"  // generated by GHC
#include "Zap_stub.h"

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     ){
  return TRUE;
}

extern "C"{

LEWIS_API HsBool lewis_Begin(){
  int argc = ...
  char *argv[] = ...

  // Initialize Haskell runtime
  hs_init(&argc, &argv);

  // Tell Haskell about all root modules
  hs_add_root(__stginit_Bar);

  // do any other initialization here and
  // return false if there was a problem
  return HS_BOOL_TRUE;
}

LEWIS_API void lewis_End(){
  hs_exit();
}

LEWIS_API HsInt lewis_Test(HsInt x){
  // use Haskell functions exported by
  // modules Bar and/or Zap
```

```
  return ...
 }

 } // extern "C"

and some application which used the functions in the DLL would have a main() function like:

 // MyApp.cpp
 #include "stdafx.h"
 #include "Lewis.h"

 int main(int argc, char *argv[]){
   if (lewis_Begin()){
      // can now safely call other functions
      // exported by Lewis DLL

   }
   lewis_End();
   return 0;
 }
```

`Lewis.h` would have to have some appropriate `#ifndef` to ensure that the Haskell FFI types were defined for external users of the DLL (who wouldn't necessarily have GHC installed and therefore wouldn't have the include files like `HsFFI.h` etc).

Chapter 12

Known bugs and infelicities

12.1 Haskell 98 vs. Glasgow Haskell: language non-compliance

This section lists Glasgow Haskell infelicities in its implementation of Haskell 98. See also the "when things go wrong" section (Chapter 9) for information about crashes, space leaks, and other undesirable phenomena.

The limitations here are listed in Haskell Report order (roughly).

12.1.1 Divergence from Haskell 98

12.1.1.1 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 `.\`.

12.1.1.2 Context-free syntax

- 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
```

- GHC doesn't do fixity resolution in expressions during parsing. For example, according to the Haskell report, the following expression is legal Haskell:

```
    let x = 42 in x == 42 == True
```

  and parses as:

```
    (let x = 42 in x == 42) == True
```

  because according to the report, the `let` expression 'extends as far to the right as possible'. Since it can't extend past the second equals sign without causing a parse error (`==` is non-fix), the `let`-expression must terminate there. GHC simply gobbles up the whole expression, parsing like this:

```
(let x = 42 in x == 42 == True)
```

The Haskell report is arguably wrong here, but nevertheless it's a difference between GHC & Haskell 98.

12.1.1.3 Expressions and patterns

None known.

12.1.1.4 Declarations and bindings

GHC's typechecker makes all pattern bindings monomorphic by default; this behaviour can be disabled with `-XNoMonoPatBinds`. See Section 7.1.

12.1.1.5 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 4.6.9. 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."

12.1.1.6 Numbers, basic types, and built-in classes

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.

12.1.1.7 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.

`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`.

Strings treated as ISO-8859-1 Various library functions, such as `putStrLn`, treat Strings as if they were ISO-8859-1 rather than UTF-8.

12.1.2 GHC's interpretation of undefined behaviour in Haskell 98

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).

12.1.3 Divergence from the FFI specification

`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.

12.2 Known bugs or infelicities

The bug tracker lists bugs that have been reported in GHC but not yet fixed: see the SourceForge GHC page. 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.

12.2.1 Bugs in GHC

- GHC can warn about non-exhaustive or overlapping patterns (see Section 4.7), 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.

- GHC does not keep careful track of what instance declarations are 'in scope' if they come from other packages. Instead, all instance declarations that GHC has seen in other packages are all in scope everywhere, whether or not the module from that package is used by the command-line expression. This bug affects only the `--make` mode and GHCi.

12.2.2 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 13

Index