The Glasgow Haskell Compiler User's Guide, Version 5.04

The GHC Team

The Glasgow Haskell Compiler User's Guide, Version 5.04

by The GHC Team

Table of 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 (http://www.haskell.org/) language.

GHC has two main components: an interactive Haskell interpreter (also known as GHCi), described in Chapter 3, 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 large collection of libraries, with everything from parser combinators to networking. The libraries are described in separate documentation.

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

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

- Haskell home page (http://www.haskell.org/)
- GHC home page (http://www.haskell.org/ghc/)
- comp.lang.functional FAQ (http://www.cs.nott.ac.uk/Department/Staff/mpj/faq.html)

We run the following mailing lists about Glasgow Haskell. 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 (Chapter 13).

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:

Send bug reports for GHC to this address! The sad and lonely people who subscribe to this list will muse upon what's wrong and what you might do about it.

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 CVS repository. There are several other similar lists for other parts of the CVS repository (eg. `cvs-hslibs`, `cvs-happy`, `cvs-hdirect` etc.)

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.

1.2. Reporting bugs in GHC

Glasgow Haskell is a changing system so there are sure to be bugs in it.

To report a bug, either:

- Go to the SoureForge GHC page (http://sourceforge.net/projects/ghc/), go to the "bugs" section, click on "submit", and enter your bug report. You can also check the outstanding bugs here and search the archives to make sure it hasn't already been reported. Or:
- Email your bug report to <glasgow-haskell-bugs@haskell.org>.

1.2.1. How do I tell if I should report my bug?

Take a look at the FAQ (Chapter 13) and Chapter 9, which will give you some guidance as to whether the behaviour you're seeing is really a bug or not.

If it is a bug, then it might have been reported before: try searching the mailing list archives. The archives don't have a built-in search facility, but we find that Google (http://www.google.com/)'s site search works pretty well: enter "site:www.haskell.org" followed by your search term into Google.

If in doubt, just report it.

1.2.2. What to put in a bug report

The name of the bug-reporting game is: facts, facts, facts. Don't omit them because "Oh, they won't be interested…"

1. What kind of machine are you running on, and exactly what version of the operating system are you using? (on a Unix system, **uname -a** or **cat /etc/motd** will show the desired information.)

2. What version of GCC are you using? **gcc -v** will tell you.
3. Run the sequence of compiles/runs that caused the offending behaviour, capturing all the input/output in a "script" (a UNIX command) or in an Emacs shell window. We'd prefer to see the whole thing.
4. Be sure any Haskell compilations are run with a `-v` (verbose) flag, so we can see exactly what was run, what versions of things you have, etc.
5. What is the program behaviour that is wrong, in your opinion?
6. If practical, please send enough source files for us to duplicate the problem.
7. If you are a Hero and track down the problem in the compilation-system sources, please send us patches relative to a known released version of GHC, or whole files if you prefer.

1.3. GHC version numbering policy

As of GHC version 4.08, we have adopted the following policy for numbering GHC versions:

Stable Releases

These are numbered `x.yy.z`, where `yy` is *even*, and `z` is the patchlevel number (the trailing `.z` can be omitted if `z` is zero). Patchlevels are bug-fix releases only, and never change the programmer interface to any system-supplied code. However, if you install a new patchlevel over an old one you will need to recompile any code that was compiled against the old libraries.

The value of `__GLASGOW_HASKELL__` (see Section 4.12.3) for a major release `x.yy.z` is the integer `xyy`.

Snapshots/unstable releases

We may make snapshot releases of the current development sources from time to time, and the current sources are always available via the CVS repository (see the GHC web site for details).

Snapshot releases are named `x.yy.YYYYMMDD` where `yy` is *odd*, and `YYYYMMDD` is the date of the sources from which the snapshot was built. In theory, you can check out the exact same sources from the CVS repository using this date.

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 4.4).

1.4. Release notes for version 5.04

1.4.1. User-visible compiler changes

- Full support for MacOS X, including fully optimized compilation, has been added. Only a native code generator and support for `-split-objs` is still missing. Everything else needs more testing, but should work.
- `ghc-pkg`: new options `--auto-ghci-libs`, `-u`/`--update-package`, `--force`, and `-i`/`--input-file`, and suppport for expanding environment variables in package descriptions. See Section 4.10).
- The latest version of the FFI spec is fully supported. The syntax of FFI declarations has changed accordingly. The old syntax is still accepted for the time being, but will elicit a warning from the compiler.
- New option: `-F` specifies a user-defined preprocessing phase (see Section 4.12.4).
- Major overhaul of the heap profiling subsystem, with new facilities for retainer profiling and biographical profiling (ala nhc98, albeit with a couple of omissions). The syntax of the runtime heap-profiling options has changed. See Section 5.4.
- The type system now supports full rank-N types (previously only limited rank-2 types were supported). See Section 7.3.9.
- Explicit kind annotations can now be given on any binding occurrence of a type variable. See Section 7.3.3.
- The handling of type synonyms has been rationalised. See Section 7.3.10.
- Fixes for several space leaks in the compiler itself (these fixes were also merged into 5.02.3).
- It is now possible to derive arbitrary classes for newtypes. See Section 7.9.
- Deadlock is now an exception, rather than a return status from the scheduler. See the module `Control.Exception` in the library documentation for more details.
- The syntax and behaviour of `RULE` pragmas has changed slightly. See Section 7.7.
- Interface files are now in a binary format to reduce compilation times. To view an interface file in plain text, use the `--show-iface` flag.
- A restriction on the form of class declarations has been lifted. In Haskell 98, it is illegal for class method types to mention constraints on the class type variable. eg.

```
class Seq s a where
```

```
elem     :: Eq a => a -> s a -> Bool
```

This restriction has now been lifted in GHC.

- Main threads can now receive the `BlockedOnDeadMVar` exception in the same way as other threads.
- The `-fall-strict` flag never really worked, and has been removed.
- The syntax of `.hi-boot` files is now much clearer and Haskell-like. See Section 4.9.7.
- There is a new flag `-fffi` which enables FFI support without turning on the rest of the GHC extensions.
- The syntax for implicit parameter bindings has changed. Previously the keyword `with` was used to introduce implicit bindings, but now implicit bindings may be introduced using `let` (see Section 7.3.6). As a result of this, `with` is no longer a keyword when `-fglasgow-exts` is turned on.

  The option `-fwith` may be used to restore the old behaviour.

- Infix type constructors are now allowed, and must begin with a colon (as with data constructors). See Section 7.3.2.
- The `do`-notation syntax is now rebindable in the same way as other built-in syntax. See Section 7.5.4.
- Support for using "frameworks" on Darwin/MacOS X has been added. See the `-framework` option in Section 4.12.7, and the `framework_dirs` field of a package spec in Section 4.10.4.

1.4.2. User-visible interpreter (GHCi) changes

- New commands: `:browse`, `:set args`, `:set prog`, `:show binadings`, and `:show modules` (see Section 3.6).
- There is a much more flexible mechanism for manipulating the scope for expressions typed at the prompt. For example, one can now have both the `Prelude` and the exports of several compiled modules in scope at the same time. See Section 3.4.1.
- GHCi now supports `foreign import "wrapper"` FFI declarations.

1.4.3. User-visible library changes

- GHC is in the process of moving to a new hierarchical set of libraries. At the moment, we have two sets of libraries, both described in accompanying documents:

- The “new libraries” which are hierarchical and consist of the following packages: `base`, `haskell98`, `haskell-src`, and `network`. Broadly speaking, `base` contains the `Prelude`, standard libraries and most of the contents of the old `lang` package. By default, the `base` and `haskell98` packages are enabled.
- The `hslibs`, most of which are now deprecated. Where possible, new code should be written to use the new libraries instead.

  The following libraries in `hslibs` have not moved yet:

  - The packages `win32`, `xlib`, `graphics`, and `posix`.
  - The Edison libraries in the `data` package.
  - In the `lang` package, the modules `TimeExts`, `DirectoryExts`, `SystemExts`, and `NumExts`.
  - The HaXml libraries in the `text` package.
  - In the `util` package, the modules `MD5`, `Select`, `Memo`, `Observe`, and `Readline`.

  All other libraries from `hslibs` either have equivalents in the new libraries (see the `hslibs` docs for details), or were already deprecated and hence were not moved into the new hierarchy.

- The `Read` class is now based on a parsing combinator library which is vastly more efficient than the previous one. See the modules `Text.Read`. `Text.ParserCombinators.ReadP`, and `Text.ParserCombinators.ReadPrec` in the library documentation.

  The code generated by the compiler for derived `Read` instances should be much shorter than before.

1.4.4. New experimental features

- Linear implicit parameters. See Section 7.3.7.
- The RTS has support for running in a multi-threaded environment and making non-blocking (from Haskell’s point of view) calls to foreign C functions which would normally block. To enable this behaviour, configure with the `--enable-threaded-rts` option.
- The compiler can now read in files containing Core syntax (such as those produced by the `-fext-core` option) and compile them. Input files with the `.hcr` file extension are assumed to contain Core syntax.

1.4.5. Internal changes

- Happy 1.13 is now required to build GHC, because of the change in names of certain libraries.

Chapter 2. Installing GHC

Installing from binary distributions is easiest, and recommended! (Why binaries? Because GHC is a Haskell compiler written in Haskell, so you've got to bootstrap it somehow. We provide machine-generated C-files-from-Haskell for this purpose, but it's really quite a pain to use them. If you must build GHC from its sources, using a binary-distributed GHC to do so is a sensible way to proceed. For the other `fptools` programs, many are written in Haskell, so binary distributions allow you to install them without having a Haskell compiler.)

This guide is in several parts:

- Installing on Unix-a-likes (Section 2.1).
- Installing on Windows (Section 2.2).
- The layout of installed files (Section 2.3). You don't need to know this to install GHC, but it's useful if you are changing the implementation.

2.1. Installing on Unix-a-likes

2.1.1. When a platform-specific package is available

For certain platforms, we provide GHC binaries packaged using the native package format for the platform. This is likely to be by far the best way to install GHC for your platform if one of these packages is available, since dependencies will automatically be handled and the package system normally provides a way to uninstall the package at a later date.

We generally provide the following packages:

RedHat or SuSE Linux/x86

RPM source & binary packages for RedHat and SuSE Linux (x86 only) are available for most major releases.

Debian Linux/x86

Debian packages for Linux (x86 only), also for most major releases.

FreeBSD/x86

On FreeBSD/x86, GHC can be installed using either the ports tree (`cd /usr/ports/lang/ghc && make install`) or from a pre-compiled package available from your local FreeBSD mirror.

Other platform-specific packages may be available, check the GHC download page for details.

2.1.2. GHC binary distributions

Binary distributions come in "bundles," one bundle per file called `bundle-platform.tar.gz`. (See the building guide for the definition of a platform.) Suppose that you untar a binary-distribution bundle, thus:

```
% cd /your/scratch/space
% gunzip < ghc-x.xx-sun-sparc-solaris2.tar.gz | tar xvf -
```

Then you should find a single directory, `ghc-version`, with the following structure:

`Makefile.in`

the raw material from which the `Makefile` will be made (Section 2.1.2.1).

`configure`

the configuration script (Section 2.1.2.1).

`README`

Contains this file summary.

`INSTALL`

Contains this description of how to install the bundle.

`ANNOUNCE`

The announcement message for the bundle.

`NEWS`

release notes for the bundle—a longer version of `ANNOUNCE`. For GHC, the release notes are contained in the User Guide and this file isn't present.

`bin/platform`

contains platform-specific executable files to be invoked directly by the user. These are the files that must end up in your path.

`lib/platform/`

contains platform-specific support files for the installation. Typically there is a subdirectory for each `fptools` project, whose name is the name of the project with its version number. For example, for GHC there would be a sub-directory `ghc-x.xx/` where `x.xx` is the version number of GHC in the bundle.

These sub-directories have the following general structure:

`libHSstd.a` etc:

supporting library archives.

`ghc-iface.prl` etc:

support scripts.

`import/`

(`.hi`) for the prelude.

`include/`

A few C `#include` files.

`share/`

contains platform-independent support files for the installation. Again, there is a sub-directory for each `fptools` project.

`html/`

contains HTML documentation files (one sub-directory per project).

`man/`

contains Unix manual pages.

2.1.2.1. Installing

OK, so let's assume that you have unpacked your chosen bundles into a scratch directory `fptools`. What next? Well, you will at least need to run the `configure` script by changing your directory to `fptools` and typing `./configure`. That should convert `Makefile.in` to `Makefile`.

You can now either start using the tools *in-situ* without going through any installation process, just type `make in-place` to set the tools up for this. You'll also want to add the path which `make` will now echo to your `PATH` environment variable. This option is useful if you simply want to try out the package and/or you don't have the necessary privileges (or inclination) to properly install the tools locally. Note that if you do decide to install the package 'properly' at a later date, you have to go through the installation steps that follow.

To install an `fptools` package, you'll have to do the following:

1. Edit the `Makefile` and check the settings of the following variables:

`platform`

the platform you are going to install for.

`bindir`

the directory in which to install user-invokable binaries.

`libdir`

the directory in which to install platform-dependent support files.

`datadir`

the directory in which to install platform-independent support files.

`infodir`

the directory in which to install Emacs info files.

`htmldir`

the directory in which to install HTML documentation.

`dvidir`

the directory in which to install DVI documentation.

The values for these variables can be set through invocation of the **configure** script that comes with the distribution, but doing an optical diff to see if the values match your expectations is always a Good Idea.

Instead of running ***configure****, it is perfectly OK to copy* `Makefile.in` *to* `Makefile` *and set all these variables directly yourself. But do it right!*

2. Run `make install`. This *should* work with ordinary Unix `make`—no need for fancy stuff like GNU `make`.
3. `rehash` (t?csh or zsh users), so your shell will see the new stuff in your bin directory.
4. Once done, test your "installation" as suggested in Section 2.1.2.3. Be sure to use a `-v` option, so you can see exactly what pathnames it's using. If things don't work as expected, check the list of known pitfalls in the building guide.

When installing the user-invokable binaries, this installation procedure will install GHC as `ghc-x.xx` where `x.xx` is the version number of GHC. It will also make a link (in the binary installation directory) from `ghc` to `ghc-x.xx`. If you install multiple versions of GHC then the last one "wins", and "`ghc`" will invoke the last one installed. You can change this manually if you want. But regardless, `ghc-x.xx` should always invoke GHC version `x.xx`.

2.1.2.2. What bundles there are

There are plenty of "non-basic" GHC bundles. The files for them are called `ghc-x.xx-bundle-platform.tar.gz`, where the *platform* is as above, and *bundle* is one of these:

`prof`:

Profiling with cost-centres. You probably want this.

`par`:

Parallel Haskell features (sits on top of PVM). You'll want this if you're into that kind of thing.

`gran`:

The "GranSim" parallel-Haskell simulator (hmm… mainly for implementors).

`ticky`:

"Ticky-ticky" profiling; very detailed information about "what happened when I ran this program"—really for implementors.

One likely scenario is that you will grab *two* binary bundles—basic, and profiling. We don't usually make the rest, although you can build them yourself from a source distribution.

The various GHC bundles are designed to be unpacked into the same directory; then installing as per the directions above will install the whole lot in one go. Note: you *must* at least have the basic GHC binary distribution bundle, these extra bundles won't install on their own.

2.1.2.3. Testing that GHC seems to be working

The way to do this is, of course, to compile and run *this* program (in a file `Main.hs`):

```
main = putStr "Hello, world!\n"
```

Compile the program, using the `-v` (verbose) flag to verify that libraries, etc., are being found properly:

```
% ghc -v -o hello Main.hs
```

Now run it:

```
% ./hello
Hello, world!
```

Some simple-but-profitable tests are to compile and run the notorious `nfib` program, using different numeric types. Start with `nfib :: Int -> Int`, and then try `Integer`, `Float`, `Double`, `Rational` and perhaps the overloaded version. Code for this is distributed in `ghc/misc/examples/nfib/` in a source distribution.

For more information on how to "drive" GHC, read on...

2.2. Installing on Windows

Getting the Glasgow Haskell Compiler (post 5.02) to run on Windows platforms is a snap: the Installshield does everything you need.

2.2.1. Installing GHC on Windows

To install GHC, use the following steps:

- Download the Installshield `setup.exe` from the GHC download page haskell.org (http://www.haskell.org/ghc).
- Run `setup.exe`. (If you have a previous version of GHC, Installshield will offer to "modify", or "remove" GHC. Choose "remove"; then run `setup.exe` a second time. This time it should offer to install.)

  At this point you should find GHCi and the GHC documentation are available in your Start menu under "Start/Programs/Glasgow Haskell Compiler".

- The final dialogue box from the install process tells you where GHC has been installed. If you want to invoke GHC from a command line, add this to your PATH environment variable. Usually, GHC installs into `c:/ghc/ghc-5.02`, though the last part of this path depends on which version of GHC you are installing, of course. You need to add `c:/ghc/ghc-5.02/bin` to your path if yo
- GHC needs a directory in which to create, and later delete, temporary files. It uses the standard Windows procedure `GetTempPath()` to find a suitable directory. This procedure returns:
  - The path in environment variable TMP, if TMP is set.
  - Otherwise, the path in environment variable TEMP, if TEMP is set.
  - Otherwise, there is a per-user default which varies between versions of Windows. On NT and XP-ish versions, it might be: `c:\Documents and Settings\<username>\Local Settings\Temp`

The main point is that if you don't do anything GHC will work fine; but if you want to control where the directory is, you can do so by setting TMP or TEMP.

- To test the fruits of your labour, try now to compile a simple Haskell program:

```
bash$ cat main.hs
module Main(main) where

main = putStrLn "Hello, world!"
bash$ ghc -o main main.hs
..
bash$ ./main
Hello, world!
bash$
```

You do *not* need the Cygwin toolchain, or anything else, to install and run GHC.

An installation of GHC requires about 140M of disk space. To run GHC comfortably, your machine should have at least 64M of memory.

2.2.2. Moving GHC around

At the moment, GHC installs in a fixed place (`c:/ghc/ghc-x.yy`, but once it is installed, you can freely move the entire GHC tree just by copying the `ghc-x.yy` directory. (You may need to fix up the links in "Start/Programs/Glasgow Haskell Compiler" if you do this.)

It is OK to put GHC tree in a directory whose path involves spaces. However, don't do this if you use want to use GHC with the Cygwin tools, because Cygwin can get confused when this happpens. We havn't quite got to the bottom of this, but so far as we know it's not a problem with GHC itself. Nevertheless, just to keep life simple we usually put GHC in a place with a space-free path.

2.2.3. Installing ghc-win32 FAQ

1. I'm having trouble with symlinks.

Symlinks only work under Cygwin (Section 2.1.2.1), so binaries not linked to the Cygwin DLL, in particular those built for Mingwin, will not work with symlinks.

2. I'm getting "permission denied" messages from the **rm** or **mv**.

This can have various causes: trying to rename a directory when an Explorer window is open on it tends to fail. Closing the window generally cures the problem, but sometimes its cause is more mysterious, and logging off and back on or rebooting may be the quickest cure.

2.3. The layout of installed files

This section describes what files get installed where. You don't need to know it if you are simply installing GHC, but it is vital information if you are changing the implementation.

GHC is installed in two directory trees:

Binary directory

: known as `$(bindir)`, holds executables that the user is expected to invoke. Notably, `ghc` and `ghci`. On Unix, this directory is typically something like `/usr/local/bin`. On Windows, however, this directory is always `$(libdir)/bin`.

Library directory,

: known as `$(libdir)`, holds all the support files needed to run GHC. On Unix, this directory is usually something like `/usr/lib/ghc/ghc-5.02`.

When GHC runs, it must know where its library directory is. It finds this out in one of two ways:

- `$(libdir)` is passed to GHC using the `-B` flag. On Unix (but not Windows), the installed `ghc` is just a one-line shell script that invokes the real GHC, passing a suitable `-B` flag. [All the user-supplied flags follow, and a later `-B` flag overrides an earlier one, so a user-supplied one wins.]
- On Windows (but not Unix), if no `-B` flag is given, GHC uses a system call to find the directory in which the running GHC executable lives, and derives `$(libdir)` from that. [Unix lacks such a system call.]

2.3.1. Layout of the library directory

The layout of the library directory is almost identical on Windows and Unix, as follows: layout:

```
$(libdir)/
  package.conf           GHC package configuration
  ghc-usage.txt          Message displayed by ghc --help

  bin/                   [Win32 only]  User-visible binaries
       ghc.exe
       ghci.bat

  unlit                  Remove literate markup

  touchy.exe             [Win32 only]
  perl.exe               [Win32 only]
  gcc.exe                [Win32 only]

  ghc-x.xx               GHC executable [Unix only]
```

```
    ghc-split               Asm code splitter
    ghc-asm             Asm code mangler

    gcc-lib/                [Win32 only] Support files for gcc
        specs               gcc configuration

        cpp0.exe            gcc support binaries
        as.exe
        ld.exe

        crt0.o               Standard
    ..etc..     binaries

        libmingw32.a         Standard
    ..etc..     libraries

        *.h                  Include files

    imports/                 GHC interface files
        std/*.hi               'std' library
 lang/*.hi              'lang' library
        ..etc..

    include/                  C header files
        StgMacros.h            GHC-specific
        ..etc...               header files

        mingw/*.h             [Win32 only] Mingwin header files

    libHSrts.a               GHC library archives
    libHSstd.a
    libHSlang.a
      ..etc..

    HSstd1.o                 GHC library linkables
    HSstd2.o                   (used by ghci, which does
    HSlang.o                   not grok .a files yet)
```

Note that:

- On Win32, the `$(libdir)/bin` directory contains user-visible binaries; add it to your `PATH`. The `ghci` executable is a `.bat` file which invokes `ghc`.

  The GHC executable is the Real Thing (no intervening shell scripts or `.bat` files). Reason: we sometimes invoke GHC with very long command lines, and `cmd.exe` (which executes `.bat` files) truncates them. [We assume people won't invoke ghci with very long command lines.]

  On Unix, the user-invokable `ghc` invokes `$(libdir)/ghc-`*`version`*, passing a suitable `-B` flag.

- `$(libdir)` also contains support binaries. These are *not* expected to be on the user's `PATH`, but and are invoked directly by GHC. In the Makefile system, this directory is also called `$(libexecdir)`, but *you are not free to change it*. It must be the same as `$(libdir)`.
- We distribute `gcc` with the Win32 distribution of GHC, so that users don't need to install `gcc`, nor need to care about which version it is. All `gcc`'s support files are kept in `$(libdir)/gcc-lib/`.
- Similarly, we distribute `perl` and a `touch` replacement (`touchy.exe`) with the Win32 distribution of GHC.
- The support programs `ghc-split` and `ghc-asm` are Perl scripts. The first line says `#!/bin/perl`; on Unix, the script is indeed invoked as a shell script, which invokes Perl; on Windows, GHC invokes `$(libdir)/perl.exe` directly, which treats the `#!/bin/perl` as a comment. Reason: on Windows we want to invoke the Perl distributed with GHC, rather than assume some installed one.

Chapter 3. 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 famililar with Hugs (http://www.haskell.org/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.

3.1. Introduction to GHCi

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

```
$ ghci
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 5.04, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Loading package haskell98 ... linking ... done.
Prelude>
```

There may be a short pause while GHCi loads the prelude and standard libraries, after which the prompt is shown. If we follow the instructions and type `:?` for help, we get:

```
 Commands available from the prompt:

   <stmt>                     evaluate/run <stmt>
   :add <filename> ...        add module(s) to the current target set
   :browse [*]<module>        display the names defined by <module>
   :cd <dir>                  change directory to <dir>
   :def <cmd> <expr>          define a command :<cmd>
   :help, :?                  display this list of commands
   :info [<name> ...]         display information about the given names
   :load <filename> ...       load module(s) and their dependents
   :module [+/-] [*]<mod> ... set the context for expression evaluation
   :reload                    reload the current module set

   :set <option> ...          set options
   :set args <arg> ...        set the arguments returned by System.getArgs
   :set prog <progname>       set the value returned by System.getProgName

   :show modules              show the currently loaded modules
   :show bindings             show the current bindings made at the prompt

   :type <expr>               show the type of <expr>
```

```
   :undef <cmd>                undefine user-defined command :<cmd>
   :unset <option> ...         unset options
   :quit                       exit GHCi
   :!<command>                 run the shell command <command>

 Options for ':set' and ':unset':

    +r                 revert top-level expressions after each evaluation
    +s                 print timing/memory stats after each evaluation
    +t                 print type after evaluation
    -<flags>           most GHC command line flags can also be set here
                         (eg. -v2, -fglasgow-exts, etc.)
```

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.

3.2. Loading source files

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

```
main = print (fac 20)

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

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

```
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 3.4.1). 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.

3.2.1. Modules vs. filenames

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

There is one exception to this general rule: when you load a program with `:load`, 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 -idir_1:...:dir_n

or it can be set using the `:set` command from within GHCi (see Section 3.7.2)[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.

One final note: if you load a module called Main, it must contain a `main` function, just like in GHC.

3.2.2. Making changes and recompilation

If you make some changes to the source code and want GHCi to recompile the program, give the `:reload` 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 4.9.5).

3.3. Loading compiled code

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

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

When loading up source files with `:load`, GHCi 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
Skipping  D                ( D.hs, D.o )
Compiling C                ( C.hs, interpreted )
Compiling B                ( B.hs, interpreted )
Compiling A                ( A.hs, interpreted )
Ok, modules loaded: A, B, C, D.
*Main>
```

In the messages from the compiler, we see that it skipped D, and used the object file `D.o`. The message `Skipping` *`module`* indicates that compilation for *`module`* isn't necessary, 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 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 )
Skipping  C                 ( C.hs, interpreted )
Skipping  B                 ( B.hs, interpreted )
Skipping  A                 ( A.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 C                 ( C.hs, interpreted )
Compiling B                 ( B.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
Skipping  D                 ( D.hs, D.o )
Skipping  C                 ( C.hs, C.o )
Compiling B                 ( B.hs, interpreted )
Compiling A                 ( A.hs, interpreted )
Ok, modules loaded: A, B, C, D.
```

HINT: since GHCi will only use a compiled object file if it can 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 new modules will be interpreted, but the rest of the project will remain compiled.

3.4. Interactive evaluation at the prompt

When you type an expression at the prompt, GHCi immediately evaluates and prints the result. But that's not the whole story: if you type something of type `IO a` for some `a`, then GHCi *executes* it as an IO-computation, and doesn't attempt to print the result:.

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

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
No instance for `Show (a -> a)'
arising from use of `print'
in a `do' expression pattern binding: print it
```

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

3.4.1. 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. For technical reasons, GHCi can only support the `*`-form for modules which are interpreted, so compiled modules and package modules can only contribute their exports to the current scope.

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: `:module` can be shortened to `:m`). The full syntax of the `:module` command is:

`:module [+|-] [*]`mod_1 `... [*]`mod_n

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.

3.4.1.1. 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.

3.4.2. Using do-notation at the prompt

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

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

Here's an example:

```
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.

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

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

An 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>
```

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]
```

3.4.3. 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
```

This is a result of the translation mentioned earlier, namely that an expression *e* is translated to

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

before execution, resulting in a binding for `it`.

If the expression was 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
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.

3.5. 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 3.6). 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; for example, GHCi doesn't generate interface files, so options related to interface file generation won't have any effect.

3.5.1. Packages

GHCi can make use of all the packages that come with GHC, For example, to start up GHCi with the `network` package loaded:

```
$ ghci -package network
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 5.04, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Loading package haskell98 ... linking ... done.
Loading package network ... linking ... done.
Prelude>
```

Note that GHCi will also automatically load any packages on which the requested package depends.

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.

3.5.2. Extra libraries

Extra libraries may be specified on the command line using the normal `-llib` option. For example, to load the "m" library:

```
$ ghci -lm
```

On systems with `.so`-style shared libraries, the actual library loaded will the `lib`*`lib`*`.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 overriden 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.

3.6. 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, as long as the abbreviation is not ambiguous. All of the builtin commands, with the exception of `:unset` and `:undef`, may be abbreviated to a single letter.

`:add` *`module`* ...

Add *`module`*(s) to the current *target set*, and perform a reload.

`: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 the `*` symbol is placed before the module name, then *all* the identifiers defined 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.

`: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`.

`:def` *`name expr`*

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")
```

`:help`
`:?`

Displays a list of the available commands.

`: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.

`: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.

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.

`:module [+|-] [*]`*mod*$_1$ `...` `[*]`*mod*$_n$

Sets or modifies the current context for statements typed at the prompt. See Section 3.4.1 for more details.

`:quit`

Quits GHCi. You can also quit by typing a 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 3.7 for a list of available options. The `:set` command by itself shows which options are currently set.

`:set args` *arg* ...

Sets the list of arguments which are returned when the program calls `System.getArgs`.

`:set prog` *prog*

Sets the string to be returned when the program calls `System.getProgName`.

`:show bindings`

Show the bindings made at the prompt and their types.

`:show modules`

Show the list of modules currently load.

`: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 3.7 for a list of available options.

`:!` *command*...

Executes the shell command *command*.

3.7. 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.

3.7.1. 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`'.

3.7.2. 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 4.19), may be set using `:set`. To unset an option, you can set the reverse option:

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

Section 4.19 lists the reverse for each option where applicable.

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

3.8. The `.ghci` file

When it starts, GHCi always reads and executes commands from `$HOME/.ghci`, followed by `./.ghci`.

The `.ghci` in your home directory 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.)

Two command-line options control whether the `.ghci` files are read:

`-ignore-dot-ghci`

Don't read either `./.ghci` or `$HOME/.ghci` when starting up.

`-read-dot-ghci`

Read `.ghci` and `$HOME/.ghci`. This is normally the default, but the `-read-dot-ghci` option may be used to override a previous `-ignore-dot-ghci` option.

3.9. FAQ and Things To Watch Out For

GHCi complains about `main` not being in scope when I load a module.

You probably omitted the `module` declaration at the top of the module, which causes the module name to default to `Main`. In Haskell, the `Main` module must define a function called `main`. Admittedly this doesn't make a great deal of sense for an interpreter, but the rule was kept for compatibility with GHC.

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.

No, they don't. This is because the Haskell binding to the GNU readline library doesn't support reading from the terminal in a non-blocking way, which is required to work properly with GHC's concurrency model.

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).

Notes

1. The 'i' stands for "Interactive"
2. except `foreign export`, at the moment
3. If you started up GHCi from the command line then GHCi's current directory is the same as the current directory of the shell from which it was started. If you started GHCi from the "Start" menu in Windows, then the current directory is probably something like `C:\Documents and Settings\`*`user name`*.
4. Note that in GHCi, and `--make` mode, the `-i` option is used to specify the search path for *source* files, whereas in standard batch-compilation mode the `-i` option is used to specify the search

path for interface files, see Section 4.9.2.

Chapter 4. Using GHC

GHC can work in one of three “modes”:

ghc —interactive

Interactive mode, which is also available as **ghci**. Interactive mode is described in more detail in Chapter 3.

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**.

ghc [-E | -C | -S | -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.

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` pragma :

```
{-# OPTIONS -fglasgow-exts #-}
module X where
...
```

`OPTIONS` pragmas are only looked for at the top of your source files, upto the first (non-literate,non-empty) line not containing `OPTIONS`. Multiple `OPTIONS` pragmas are recognised. 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 driver maintains internally, so you'll be desperately disappointed if you try to glob etc. inside `OPTIONS`.

NOTE: the contents of OPTIONS are prepended to the command-line options, so you *do* have the ability to override OPTIONS settings via 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` pragma is the Right Thing. (If you use `-keep-hc-file-too` and have OPTION flags in your module, the OPTIONS 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 3.7 for more details.

4.2. Static vs. Dynamic options

Each of GHC's command line options is classified as either *static* or *dynamic*. A static flag may only be specified on the command line, whereas a dynamic flag may also be given in an `OPTIONS` pragma in a source file or set from the GHCi command-line with `:set`.

As a rule of thumb, all the language options are dynamic, as are the warning options and the debugging options. The rest are static, with the notable exceptions of `-v`, `-cpp`, `-fasm`, `-fvia-C`, and `-#include`. The flag reference tables (Section 4.19) lists the status of each flag.

4.3. Meaningful file suffixes

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

`.lhs`

A "literate Haskell" module.

`.hs`

A not-so-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. Help and verbosity options

`--help`
`-?`

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

`-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).

`--version`

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

`--numeric-version`

Print GHC's numeric version number only.

`--print-libdir`

Print the path to GHC's library directory. This is the top of the directory tree containing GHC's libraries, interfaces, and include files (usually something like `/usr/local/lib/ghc-5.04` on Unix). This is the value of `$libdir`in the package configuration file (see Section 4.10).

4.5. Using ghc `--make`

When given the `--make` option, GHC will build a multi-module Haskell program by following dependencies from a single root module (usually `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 `Makefile`s are:

- GHC doesn't have to be restarted for each compilation, which means it can cache information between compilations. Compiling a muli-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` pragma (see Section 4.1.2).

If the program needs to be linked with additional objects (say, some auxilliary 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 4.9.2).

4.6. GHC without `--make`

Without `--make`, 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 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`

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 4.12.6 for more details.

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

Note: The option `-E` runs just the pre-processing passes of the compiler, dumping the result in a file. Note that this differs from the previous behaviour of dumping the file to standard output.

4.7. Re-directing the compilation output(s)

`-o`

GHC’s compiled output normally goes into a `.hc`, `.o`, etc., file, depending on the last-run compilation phase. The option `-o foo` re-directs the output of that last-run phase to file `foo`.

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!

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`).

`-odir`

The `-o` option isn’t of much use if you have *several* input files... Non-interface output files are normally put in the same directory as their corresponding input file came from. You may specify that they be put in another directory using the `-odir <dir>` (the “Oh, dear” option). For example:

```
% ghc -c parse/Foo.hs parse/Bar.hs gurgle/Bumble.hs -odir `arch`
```

The output files, `Foo.o`, `Bar.o`, and `Bumble.o` would be put into a subdirectory named after the architecture of the executing machine (`sun4`, `mips`, etc). The directory must already exist; it won't be created.

Note that the `-odir` option does *not* affect where the interface files are put. 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` *`directory`*

Redirects all generated interface files into *`directory`*, instead of the default which is to place the interface file in the same directory as the source file.

`-osuf` *`suffix`*
`-hisuf` *`suffix`*
`-hcsuf` *`suffix`*

EXOTICA: 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 4.9.4).

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

The `-hisuf`/`-osuf` game is particularly useful if you want to compile a program both with and without profiling, in the same directory. You can say:

```
ghc ...
```

to get the ordinary version, and

```
ghc ... -osuf prof.o -hisuf prof.hi -prof -auto-all
```

to get the profiled version.

4.7.1. 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-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-files`

Keep intermediate `.s` files.

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

4.7.2. 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).

4.8. Warnings and sanity-checking

GHC has a number of options that select which types of non-fatal error messages, otherwise known as warnings, can be generated during compilation. By default, you get a standard set of warnings which are generally likely to indicate bugs in your program. These are: `-fwarn-overlpapping-patterns`, `-fwarn-deprecations`, `-fwarn-duplicate-exports`, `-fwarn-missing-fields`, and `-fwarn-missing-methods`. The following flags are simple ways to select standard "packages" of warnings:

`-W`:

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

`-w`:

Turns off all warnings, including the standard ones.

`-Wall`:

Turns on all warning options.

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-deprecations`:

Causes a warning to be emitted when a deprecated function or type is used. Entities can be marked as deprecated using a pragma, see Section 7.6.7.

`-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-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 be 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-misc`:

Turns on warnings for various harmless but untidy things. This currently includes: importing a type with `(..)` when the export is abstract, and listing duplicate class assertions in a qualified type.

`-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.

`-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. This 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 cyclic definition `let x = ... x ... in`.

Consequently, this option does *will* complain about cyclic recursive definitions.

`-fwarn-overlapping-patterns`:

By default, the compiler will warn you if a set of patterns are overlapping, i.e.,

```
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`.

`-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-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.

`-fwarn-unused-imports`:

Report any objects that are explicitly imported but never used.

`-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. To eliminate the warning, all unused variables can be replaced with wildcards.

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

4.9. Separate compilation

This section describes how GHC supports separate compilation.

4.9.1. Interface files

When GHC compiles a source file `A.hs` which contains a module `A`, say, it generates an object `A.o`, *and* a companion *interface file* `A.hi`. The interface file is merely there to help the compiler compile other modules in the same program. Interfaces are in a binary format, so don't try to look at one;

however you *can* see the contents of an interface file by using GHC with the `--show-iface` option (see Section 4.9.4, below).

NOTE: In general, the name of a file containing module `M` should be named `M.hs` or `M.lhs`. The only exception to this rule is module `Main`, which can be placed in any file.

The interface file for `A` contains information needed by the compiler when it compiles any module `B` that imports `A`, whether directly or indirectly. When compiling `B`, GHC will read `A.hi` to find the details that it needs to know about things defined in `A`.

The interface file may contain all sorts of things that aren't explicitly exported from `A` by the programmer. For example, even though a data type is exported abstractly, `A.hi` will contain the full data type definition. For small function definitions, `A.hi` will contain the complete definition of the function. For bigger functions, `A.hi` will contain strictness information about the function. And so on. GHC puts much more information into `.hi` files when optimisation is turned on with the `-O` flag (see Section 4.11). Without `-O` it puts in just the minimum; with `-O` it lobs in a whole pile of stuff.

`A.hi` should really be thought of as a compiler-readable version of `A.o`. If you use a `.hi` file that wasn't generated by the same compilation run that generates the `.o` file the compiler may assume all sorts of incorrect things about `A`, resulting in core dumps and other unpleasant happenings.

4.9.2. Finding interface files

In your program, you import a module `Foo` by saying `import Foo`. GHC goes looking for an interface file, `Foo.hi`. It has a builtin list of directories (notably including `.`) where it looks.

`-i<dirs>`

This flag appends a colon-separated list of `dirs` to the "import directories" list, which initially contains a single entry: ".".

This list is scanned before any package directories (see Section 4.10) when looking for imports, but note that if you have a home module with the same name as a package module then this is likely to cause trouble in other ways, with link errors being the least nasty thing that can go wrong...

See also Section 4.9.5 for the significance of using relative and absolute pathnames in the `-i` list.

`-i`

resets the "import directories" list back to nothing.

See also the section on packages (Section 4.10), which describes how to use installed libraries.

4.9.3. Finding interfaces for hierarchical modules

GHC supports a hierarchical module namespace as an extension to Haskell 98 (see Section 7.5.1).

A module name in general consists of a sequence of components separated by dots (‘`.`’). When looking for interface files for a hierarchical module, the compiler turns the dots into path separators, so for example a module `A.B.C` becomes `A/B/C` (or `A\B\C` under Windows). Then each component of the import directories list is tested in turn; so for example if the list contains directories D_1 to D_n, then the compiler will look for the interface in D_1`/A/B/C.hi` first, then D_2`/A/B/C.hi` and so on.

Note that it’s perfectly reasonable to have a module which is both a leaf and a branch of the tree. For example, if we have modules `A.B` and `A.B.C`, then `A.B`’s interface file will be in `A/B.hi` and `A.B.C`’s interface file will be in `A/B/C.hi`.

For GHCi and `--make`, the search strategy for source files is exactly the same, just replace the `.hi` suffix in the above description with `.hs` or `.lhs`.

4.9.4. 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 run **diff** on 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.

4.9.5. The recompilation checker

`-no-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 `A.hi` should force a recompilation of `C`. 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 keeps a version number on each interface file, and on each type signature within the interface file. It also keeps in every interface file a list of the version numbers 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 reompilation checking is on, GHC will be clever. It compares the version numbers on the things it needs this time with the version numbers 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 rather early in the process saying "Compilation IS NOT required". What a beautiful sight!

Patrick Sansom had a workshop paper about how all this is done (though the details have changed quite a bit). Ask him (mailto:sansom@dcs.gla.ac.uk) if you want a copy.

4.9.6. 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)

# 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 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.

4.9.6.1. 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**. **ghc -M** will append the needed dependencies to your `Makefile`.

In general, if module `A` contains the line

```
import B ...blah...
```

then **ghc -M** will generate a dependency line of the form:

```
A.o : B.hi
```

If module `A` contains the line

```
import {-# SOURCE #-} B ...blah...
```

then **ghc -M** will generate a dependency line of the form:

```
A.o : B.hi-boot
```

(See Section 4.9.7 for details of `hi-boot` style interface files.) If `A` imports multiple modules, then there will be multiple lines with `A.o` as the target.

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-prelude` option below).

The dependency generation phase of GHC can take some additional options, which you may find useful. For historical reasons, each option passed to the dependency generator from the GHC command line must be preceded by `-optdep`. For example, to pass `-f .depend` to the dependency generator, you say

```
ghc -M -optdep-f -optdep.depend ...
```

The options which affect dependency generation are:

`-w`

Turn off warnings about interface file shadowing.

`-f` *`file`*

Use *`file`* as the makefile, rather than `makefile` or `Makefile`. If *`file`* doesn't exist, **mkdependHS** creates it. We often use `-f .depend` to put the dependencies in `.depend` and then **include** the file `.depend` into `Makefile`.

`-o <osuf>`

Use `.<osuf>` as the "target file" suffix (default: `o`). Multiple `-o` flags are permitted (GHC2.05 onwards). Thus "`-o hc -o o`" will generate dependencies for `.hc` and `.o` files.

-s <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 -s flags are permitted. For example, -o hc -s a -s b will make dependencies for .hc on .hi, .a_hc on .a_hi, and .b_hc 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.

-x

same as --exclude-module

--exclude-directory=<dirs>

Regard the colon-separated list of directories <dirs> as containing stable, don't generate any dependencies on modules therein.

--include-module=<file>

Regard <file> as not "stable"; i.e., generate dependencies on it (if any). This option is normally used in conjunction with the --exclude-directory option.

--include-prelude

Regard modules imported from packages as unstable, i.e., generate dependencies on the package modules used (including Prelude, and all other standard Haskell libraries). This option is normally only used by the various system libraries.

4.9.7. How to compile mutually recursive modules

Currently, the compiler does not have proper support for dealing with mutually recursive modules:

```
module A where

import B

newtype TA = MkTA Int

f :: TB -> TA
f (MkTB x) = MkTA x
--------
module B where

import A
```

```
data TB = MkTB !Int

g :: TA -> TB
g (MkTA x) = MkTB x
```

When compiling either module A and B, the compiler will try (in vain) to look for the interface file of the other. So, to get mutually recursive modules off the ground, you need to hand write an interface file for A or B, so as to break the loop. These hand-written interface files are called `hi-boot` files, and are placed in a file called `<module>.hi-boot`. To import from an `hi-boot` file instead of the standard `.hi` file, use the following syntax in the importing module:

```
import {-# SOURCE #-} A
```

The hand-written interface 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 that imports `A` recursively.

For the example at hand, the boot interface file for A would look like the following:

```
module A where
newtype TA = MkTA GHC.Base.Int
```

The syntax is similar to a normal Haskell source file, but with some important differences:

- Non-local entities must be qualified with their *original* defining module. Qualifying by a module which just re-exports the entity won't do. In particular, most `Prelude` entities aren't actually defined in the `Prelude` (see for example `GHC.Base.Int` in the above example).
- Only `data`, `type`, `newtype`, `class`, and type signature declarations may be included.

Notice that we only put the declaration for the newtype `TA` in the `hi-boot` file, not the signature for `f`, since `f` isn't used by `B`.

If you want an `hi-boot` file to export a data type, but you don't want to give its constructors (because the constructors aren't used by the SOURCE-importing module), you can write simply:

```
module A where
data TA
```

(You must write all the type parameters, but leave out the '=' and everything that follows it.)

4.9.8. 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, 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", defined thus:

- 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 none of the type constructors or classes mentioned in the instance head (the part after the "`=>`") are declared 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.

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.

You can identify an orphan module by looking in its interface file, `M.hi`, using the `--show-iface`. If there is a "!" on the first line, GHC considers it an orphan module.

4.10. Packages

Packages are collections of libraries, conveniently grouped together as a single entity. The package system is flexible: a package may consist of Haskell code, foreign language code (eg. C libraries), or a mixture of the two. A package is a good way to group together related Haskell modules, and is essential if you intend to make the modules into a Windows DLL (see below).

Because packages can contain both Haskell and C libraries, they are also a good way to provide convenient access to a Haskell layer over a C library.

GHC comes with several packages (see the accompanying library documentation), and packages can be added to or removed from an existing GHC installation, using the supplied `ghc-pkg` tool, described in Section 4.10.4.

4.10.1. Using a package

To use a package, add the `-package` flag to the GHC command line:

`-package` *`lib`*

This option brings into scope all the modules from package *`lib`* (they still have to be imported in your Haskell source, however). It also causes the relevant libraries to be linked when linking is being done.

Some packages depend on other packages, for example the `text` package makes use of some of the modules in the `lang` package. The package system takes care of all these dependencies, so that when you say `-package text` on the command line, you automatically get `-package lang` too.

4.10.2. Maintaining a local set of packages

When GHC starts up, it automatically reads the default set of packages from a configuration file, normally named `package.conf` in your GHC installation directory.

You can load in additional package configuration files using the `-package-conf` option:

`-package-conf` *`file`*

Read in the package configuration file *`file`* in addition to the system default file. This allows the user to have a local set of packages in addition to the system-wide ones.

To create your own package configuration file, just create a new file and put the string “`[]`” in it. Packages can be added to the new configuration file using the `ghc-pkg` tool, described in Section 4.10.4.

4.10.3. Building a package from Haskell source

It takes some special considerations to build a new package:

- A package may contain several Haskell modules. A package may span many directories, or many packages may exist in a single directory. Packages may not be mutually recursive.
- A package has a name (e.g. `base`)

- The Haskell code in a package may be built into one or more archive libraries (e.g. `libHSfoo.a`), or a single DLL on Windows (e.g. `HSfoo.dll`). The restriction to a single DLL on Windows is because the package system is used to tell the compiler when it should make an inter-DLL call rather than an intra-DLL call (inter-DLL calls require an extra indirection). *Building packages as DLLs doesn't work at the moment; see Section 11.3 for the gory details.*

  Building a static library is done by using the `ar` tool, like so:

  ```
  ar cqs libHSfoo.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 be 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 4.10.4. 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
  ```

- 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:

`-package-name` *foo*

  This option is added to the command line when compiling a module that is destined to be part of package `foo`. If this flag is omitted then the default package `Main` is assumed.

Failure to use the `-package-name` option when compiling a package will result in disaster on Windows, but is relatively harmless on Unix at the moment (it will just cause a few extra dependencies in some interface files). However, bear in mind that we might add support for Unix shared libraries at some point in the future.

It is worth noting that on Windows, when each package is built as a DLL, since a reference to a DLL costs an extra indirection, intra-package references are cheaper than inter-package references. Of course, this applies to the `Main` package as well.

4.10.4. Package management

The `ghc-pkg` tool allows packages to be added or removed from a package configuration file. By default, the system-wide configuration file is used, but alternatively packages can be added, updated or removed from a user-specified configuration file using the `--config-file` option. An empty package configuration file consists of the string "`[]`".

The `ghc-pkg` program accepts the following options:

`--add-package`
`-a`

Reads package specification from the input (see below), and adds it to the database of installed packages. The package specification must be a package that isn't already installed.

`--input-file=`*`file`*
`-i` *`file`*

Read new package specifications from file *`file`*. If a value of `"-"` is given, standard input is used. If no `-i` is present on the command-line, an input file of `"-"` is assumed.

`--auto-ghci-libs`
`-g`

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.

`--config-file` *`file`*
`-f` *`file`*

Use *`file`* instead of the default package configuration file. This, in conjunction with GHC's `-package-conf` option, allows a user to have a local set of packages in addition to the system-wide installed set.

`--list-packages`
`-l`

This option displays the list of currently installed packages.

```
$ ghc-pkg --list-packages
gmp, rts, std, lang, concurrent, data, net, posix, text, util
```

Note that your GHC installation might have a slightly different set of packages installed.

The `gmp` and `rts` packages are always present, and represent the multi-precision integer and runtime system libraries respectively. The `std` package contains the Haskell prelude and standard libraries. The rest of the packages are optional libraries.

`--remove-package` *`foo`*
`-r` *`foo`*

Removes the specified package from the installed configuration.

`--update-package`

`-u`

Reads package specification from the input, and adds it to the database of installed packages. If a package with the same name is already installed, its configuration data is replaced with the new information. If the package doesn't already exist, it's added.

`--force`

Causes `ghc-pkg` to ignore missing directories and libraries when adding 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.

When modifying the configuration file *`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.

A package specification looks like this:

```
Package {
   name            = "mypkg",
   import_dirs     = ["${installdir}/imports/mypkg"],
   source_dirs     = [],
   library_dirs    = ["${installdir}"],
   hs_libraries    = ["HSmypkg" ],
   extra_libraries = ["HSmypkg_cbits"],
   include_dirs    = [],
   c_includes      = ["HsMyPkg.h"],
   package_deps    = ["text", "data"],
   extra_ghc_opts  = [],
   extra_cc_opts   = [],
   extra_ld_opts   = ["-lmy_clib"]
}
```

Components of a package specification may be specified in any order, and are:

`name`

The package's name, for use with the `-package` flag and as listed in the `--list-packages` list.

`import_dirs`

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).

`source_dirs`

A list of directories containing Haskell source files for this package. This field isn't used by GHC, but could potentially be used by an all-interpreted system like Hugs.

`library_dirs`

A list of directories containing libraries for this package.

`hs_libraries`

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`

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 `libHSstd_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`

A list of directories containing C includes for this package (maybe the empty list).

`c_includes`

A list of files to include for via-C compilations using this package. Typically this include file 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.

`package_deps`

A list of packages which this package depends on.

`extra_ghc_opts`

Extra arguments to be added to the GHC command line when this package is being used.

`extra_cc_opts`

Extra arguments to be added to the gcc command line when this package is being used (only for via-C compilations).

`extra_ld_opts`

Extra arguments to be added to the gcc command line (for linking) when this package is being used.

`framework_dirs`

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.

`extra_frameworks`

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.

The `ghc-pkg` tool performs expansion of environment variables occurring in input package specifications. So, if the `mypkg` was added to the package database as follows:

```
$ installdir=/usr/local/lib ghc-pkg -a < mypkg.pkg
```

The occurrence of `${installdir}` is replaced with `/usr/local/lib` in the package data that is added for `mypkg`.

This feature enables the distribution of package specification files that can be easily configured when installing.

For examples of more package specifications, take a look at the `package.conf` in your GHC installation.

4.11. 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.

4.11.1. `-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.

Once you choose a `-O*` "package," stick with it—don't chop and change. Modules' interfaces *will* change with a shift to a new `-O*` option, and you may have to recompile a large chunk of all importing modules before your program can again be run safely (see Section 4.9.5).

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 yet in GHC 5.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 `-O -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.

4.11.2. `-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 7.4). This flag is turned on by `-O`.

`-fno-strictness`

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

`-fno-cpr-analyse`

Turns off the CPR (constructed product result) analysis; it is somewhat experimental.

`-funbox-strict-fields`:

This option causes all constructor fields which are marked strict (i.e. “!”) to be unboxed or unpacked if possible. For example:

```
data T = T !Float !Float
```

will create a constructor `T` containing two unboxed floats if the `-funbox-strict-fields` flag is given. 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).

This option 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 !(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 !S
data S = S !Int !Int
```

will store two unboxed `Int#`s directly in the `T` constructor.

`-funfolding-update-in-place<n>`

Switches on an experimental "optimisation". Switching it on makes the compiler a little keener to inline a function that returns a constructor, if the context is that of a thunk.

```
x = plusInt a b
```

If we inlined plusInt we might get an opportunity to use update-in-place for the thunk ’x’.

`-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`).

4.12. Options related to a particular phase

4.12.1. Replacing the program for one or more phases

You may specify that a different program be used for one of the phases of the compilation system, in place of whatever the **ghc** has wired into it. For example, you might want to try a different assembler. The following options allow you to change the external program used for a given compilation phase:

`-pgmL` *`cmd`*

Use *`cmd`* as the literate pre-processor.

`-pgmP` *`cmd`*

Use *`cmd`* as the C pre-processor (with `-cpp` only).

`-pgmc` *`cmd`*

Use *`cmd`* as the C compiler.

`-pgma` *`cmd`*

Use *`cmd`* as the assembler.

`-pgml` *`cmd`*

Use *`cmd`* as the linker.

`-pgmdll` *`cmd`*

Use *`cmd`* as the DLL generator.

`-pgmdep` *`cmd`*

Use *`cmd`* as the dependency generator.

`-pgmF` *`cmd`*

Use *`cmd`* as the pre-processor (with `-F` only).

4.12.2. Forcing options to a particular phase

Options can be forced through to a particlar compilation phase, using the following flags:

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 4.16).

4.12.3. Options affecting the C pre-processor

`-cpp`

The C pre-processor **cpp** is run over your Haskell code only if the `-cpp` option is given. Unless you are building a large system with significant doses of conditional compilation, you really shouldn't need it.

`-D`*`symbol`*[=*`value`*]

Define macro *`symbol`* in the usual way. NB: does *not* affect `-D` macros passed to the C compiler when compiling via C! For those, use the `-optc-Dfoo` hack... (see Section 4.12.2).

`-U`*`symbol`*

Undefine macro *`symbol`* in the usual way.

`-I`*`dir`*

Specify a directory in which to look for `#include` files, in the usual C way.

The GHC driver pre-defines several macros when processing Haskell source code (`.hs` or `.lhs` files):

`__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 *n* of the GHC system, this will be `#define`d to *100n*. For example, for version 5.00, it is 500.

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 Chalmers.)

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 verion 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).

4.12.3.1. CPP and string gaps

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

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

don't work with `-cpp`; `/usr/bin/cpp` elides the backslash-newline pairs.

However, it appears that if you add a space at the end of the line, then **cpp** (at least GNU **cpp** and possibly other **cpp**s) leaves the backslash-space pairs alone and the string gap works as expected.

4.12.4. Options affecting a Haskell pre-processor

`-F`

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

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

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

`-pgmF` *`cmd`*

Use *`cmd`* as the Haskell pre-processor. 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 *`cmd`* 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.

4.12.5. 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. There is no real pretty way to do this, but you can use this hack from the command-line:

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

4.12.6. 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 when optimisation is off (see Section 4.11).

`-fvia-C`

Compile via C instead of using the native code generator. This is default for optimised compilations, and 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.

4.12.7. 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 `lib`*`lib`*`.a` or `lib`*`lib`*`.so` which resides somewhere on the library directories path.

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

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

`-package` *`name`*

If you are using a Haskell "package" (see Section 4.10), 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`.

`-Ldir`

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

`-framework-pathdir`

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. We use this feature for building GHC's libraries 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`

Tell the linker to use shared Haskell libraries, if available (this option is only supported on Windows at the moment, and also note that your distribution of GHC may not have been supplied with shared libraries).

`-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 driver script when linking, use `-no-hs-main`.

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.

4.13. 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` (details are in the accompanying library documentation).

RTS options are provided for modifying the behaviour of the threaded runtime system. See Section 4.14.4.

Concurrent Haskell is described in more detail in the documentation for the `Control.Concurrent` module.

4.14. Using Parallel Haskell

[You won't be able to execute parallel Haskell programs unless PVM3 (Parallel Virtual Machine, version 3) is installed at your site.]

To compile a Haskell program for parallel execution under PVM, use the `-parallel` option, both when compiling *and linking*. You will probably want to `import Parallel` into your Haskell modules.

To run your parallel program, once PVM is going, just invoke it "as normal". The main extra RTS option is `-qp<n>`, to say how many PVM "processors" your program to run on. (For more details of all relevant RTS options, please see Section 4.14.4.)

In truth, running Parallel Haskell programs and getting information out of them (e.g., parallelism profiles) is a battle with the vagaries of PVM, detailed in the following sections.

4.14.1. Dummy's guide to using PVM

Before you can run a parallel program under PVM, you must set the required environment variables (PVM's idea, not ours); something like, probably in your `.cshrc` or equivalent:

```
setenv PVM_ROOT /wherever/you/put/it
setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
setenv PVM_DPATH $PVM_ROOT/lib/pvmd
```

Creating and/or controlling your "parallel machine" is a purely-PVM business; nothing specific to Parallel Haskell. The following paragraphs describe how to configure your parallel machine interactively.

If you use parallel Haskell regularly on the same machine configuration it is a good idea to maintain a file with all machine names and to make the environment variable PVM_HOST_FILE point to this file. Then you can avoid the interactive operations described below by just saying

```
pvm $PVM_HOST_FILE
```

You use the **pvm** command to start PVM on your machine. You can then do various things to control/monitor your "parallel machine;" the most useful being:

Control-D	exit **pvm**, leaving it running
halt	kill off this "parallel machine" & exit

add <host>	add **<host>** as a processor
delete <host>	delete **<host>**
reset	kill what's going, but leave PVM up
conf	list the current configuration
ps	report processes' status
pstat <pid>	status of a particular process

The PVM documentation can tell you much, much more about **pvm**!

4.14.2. Parallelism profiles

With Parallel Haskell programs, we usually don't care about the results—only with "how parallel" it was! We want pretty pictures.

Parallelism profiles (à la **hbcpp**) can be generated with the `-qP` RTS option. The per-processor profiling info is dumped into files named `<full-path><program>.gr`. These are then munged into a PostScript picture, which you can then display. For example, to run your program `a.out` on 8 processors, then view the parallelism profile, do:

```
$ ./a.out +RTS -qP -qp8
$ grs2gr *.???.gr > temp.gr # combine the 8 .gr files into one
$ gr2ps -O temp.gr              # cvt to .ps; output in temp.ps
$ ghostview -seascape temp.ps   # look at it!
```

The scripts for processing the parallelism profiles are distributed in `ghc/utils/parallel/`.

4.14.3. Other useful info about running parallel programs

The "garbage-collection statistics" RTS options can be useful for seeing what parallel programs are doing. If you do either `+RTS -Sstderr` or `+RTS -sstderr`, then you'll get mutator, garbage-collection, etc., times on standard error. The standard error of all PE's other than the 'main thread' appears in `/tmp/pvml.nnn`, courtesy of PVM.

Whether doing `+RTS -Sstderr` or not, a handy way to watch what's happening overall is: **tail -f /tmp/pvml.nnn**.

4.14.4. RTS options for Concurrent/Parallel Haskell

Besides the usual runtime system (RTS) options (Section 4.16), there are a few options particularly for concurrent/parallel execution.

`-qp<N>`:

(PARALLEL ONLY) Use `<N>` PVM processors to run this program; the default is 2.

`-C[<us>]`:

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 milliseconds. Note that GHC's internal timer ticks every 20ms, and the context switch timer is always a multiple of this timer, so 20ms is the maximum granularity available for timed context switches.

`-q[v]`:

(PARALLEL ONLY) Produce a quasi-parallel profile of thread activity, in the file `<program>.qp`. In the style of **hbcpp**, this profile records the movement of threads between the green (runnable) and red (blocked) queues. If you specify the verbose suboption (`-qv`), the green queue is split into green (for the currently running thread only) and amber (for other runnable threads). We do not recommend that you use the verbose suboption if you are planning to use the **hbcpp** profiling tools or if you are context switching at every heap check (with `-C`). -->

`-qt<num>`:

(PARALLEL ONLY) Limit the thread pool size, i.e. the number of concurrent threads per processor to `<num>`. The default is 32. Each thread requires slightly over 1K *words* in the heap for thread state and stack objects. (For 32-bit machines, this translates to 4K bytes, and for 64-bit machines, 8K bytes.)

`-qe<num>`:

(PARALLEL ONLY) Limit the spark pool size i.e. the number of pending sparks per processor to `<num>`. The default is 100. A larger number may be appropriate if your program generates large amounts of parallelism initially.

`-qQ<num>`:

(PARALLEL ONLY) Set the size of packets transmitted between processors to `<num>`. The default is 1024 words. A larger number may be appropriate if your machine has a high communication cost relative to computation speed.

`-qh<num>`:

(PARALLEL ONLY) Select a packing scheme. Set the number of non-root thunks to pack in one packet to <num>-1 (0 means infinity). By default GUM uses full-subgraph packing, i.e. the entire subgraph with the requested closure as root is transmitted (provided it fits into one packet). Choosing a smaller value reduces the amount of pre-fetching of work done in GUM. This can be advantageous for improving data locality but it can also worsen the balance of the load in the system.

`-qg<num>`:

(PARALLEL ONLY) Select a globalisation scheme. This option affects the generation of global addresses when transferring data. Global addresses are globally unique identifiers required to maintain sharing in the distributed graph structure. Currently this is a binary option. With <num>=0 full globalisation is used (default). This means a global address is generated for every closure that is transmitted. With <num>=1 a thunk-only globalisation scheme is used, which generated global address only for thunks. The latter case may lose sharing of data but has a reduced overhead in packing graph structures and maintaining internal tables of global addresses.

4.15. Platform-specific Flags

Some flags only make sense for particular target platforms.

`-mv8`:

(SPARC machines) Means to pass the like-named option to GCC; it says to use the Version 8 SPARC instructions, notably integer multiply and divide. The similiar `-m*` GCC options for SPARC also work, actually.

`-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.

4.16. 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.

4.16.1. 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 overriden by options given on the command line.

4.16.2. 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 `-Msize`.

`-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 `-Hsize` than to increase `-Ffactor`.

The `-F` setting will be automatically reduced by the garbage collector when the maximum heap size (the `-Msize` 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).

`-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 `-Hsize`. For improving GC performance, using `-Hsize` is usually a better bet than `-Asize`.

`-ksize`

[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.

`-Ksize`

[Default: 1M] 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.

`-mn`

Minimum % `n` of heap which must be available for allocation. The default is 3%.

`-Msize`

[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.

`-s`*`file`*
`-S`*`file`*

Write modest (`-s`) or verbose (`-S`) garbage-collector statistics into file *`file`*. The default *`file`* is *`program`*`.stat`. The *`file`* `stderr` is treated specially, with the output really being sent to `stderr`.

This option is useful for watching how the storage manager adjusts the heap size based on the current amount of live data.

`-t`

Write a one-line GC stats summary after running the program. This output is in the same format as that produced by the `-Rghc-timing` option.

4.16.3. 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.

`-rfile`

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:

$< cc_1, ..., cc_n >$

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.)

4.16.4. "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 function `defaultsHook` lets you change various RTS options. The commonest 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 `-M128m -K1m`:

```
#include "Rts.h"
#include "RtsFlags.h"
void defaultsHook (void) {
   RtsFlags.GcFlags.maxStkSize  =  1000002 / sizeof(W_);
   RtsFlags.GcFlags.maxHeapSize =  128*1024*1024 / BLOCK_SIZE_W;
}
```

Don't use powers of two for heap/stack sizes: these are more likely to interact badly with direct-mapped caches. The full set of flags is defined in `ghc/rts/RtsFlags.h` the the GHC source tree.

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 ErrorHdrHook (FILE *)`

: What's printed out before the message from `error`.

`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.

`void PatErrorHdrHook (FILE *)`

: The message printed if a pattern-match fails (the failures that were not handled by the Haskell programmer).

`void PreTraceHook (FILE *)`

: What's printed out before a `trace` message.

`void PostTraceHook (FILE *)`

: What's printed out after a `trace` message.

For example, here is the "hooks" code used by GHC itself:

```
#include "Rts.h"
#include "../rts/RtsFlags.h"
#include "HsFFI.h"
```

```
void
defaultsHook (void)
{
    RtsFlags.GcFlags.heapSizeSuggestion = 6*1024*1024 / BLOCK_SIZE;
    RtsFlags.GcFlags.maxStkSize         = 8*1024*1024 / sizeof(W_);
    RtsFlags.GcFlags.giveStats = COLLECT_GC_STATS;
    RtsFlags.GcFlags.statsFile = stderr;
}

void
ErrorHdrHook (long fd)
{
    char msg[]="\n";
    write(fd,msg,1);
}

void
PatErrorHdrHook (long fd)
{
    const char msg[]="\n*** Pattern-matching error within GHC!\n\nThis is a com-
piler bug; please report it to glasgow-haskell-bugs@haskell.org.\n\nFail:";
    write(fd,msg,sizeof(msg)-1);
}

void
PreTraceHook (long fd)
{
    const char msg[]="\n";
    write(fd,msg,sizeof(msg)-1);
}

void
PostTraceHook (long fd)
{
#if 0
    const char msg[]="\n";
    write(fd,msg,sizeof(msg)-1);
#endif
}
```

4.17. Generating 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. Core files, which are given the suffix `.hcr`, can be read and processed by non-GHC back-end tools. The Core format is formally described in *An External Representation for the GHC*

Core Language (http://www.haskell.org/ghc/docs/papers/core.ps.gz), and sample tools (in Haskell) for manipulating Core files are available in the GHC source distribution directory `/fptools/ghc/utils/ext-core`. Note that the format of `.hcr` files is *different* (though similar) to the Core output format generated for debugging purposes (Section 4.18).

`-fext-core`

Generate `.hcr` files.

4.18. Debugging the compiler

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

4.18.1. Dumping out compiler intermediate structures

`-ddump-`*`pass`*

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

`-ddump-parsed`:

parser output

`-ddump-rn`:

renamer output

`-ddump-tc`:

typechecker output

`-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`:

simplifer output (Core-to-Core passes)

`-ddump-inlinings`:

inlining info from the simplifier

`-ddump-usagesp`:

UsageSP inference pre-inf and output

`-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-sat`:

output of "saturate" pass

`-ddump-stg`:

output of STG-to-STG passes

`-ddump-absC`:

*un*flattened Abstract C

`-ddump-flatC`:

flattened Abstract C

`-ddump-realC`:

same as what goes to the C compiler

`-ddump-stix`:

native-code generator intermediate form

`-ddump-asm`:

assembly language from the native-code generator

`-ddump-bcos`:

byte code compiler output

`-ddump-foreign`:

dump foreign export stubs

`-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
```

`-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). Used when even `-dverbose-simpl` doesn't cut it.

`-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.

`-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.

`-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-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.

`-dshow-unused-imports`

Have the renamer report what imports does not contribute.

4.18.2. 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).

`-dusagesp-lint`:

Turn on checks around UsageSP inference (`-fusagesp`). This verifies various simple properties of the results of the inference, and also warns if any identifier with a used-once annotation before the inference has a used-many annotation afterwards; this could indicate a non-worksafe transformation is being applied.

4.18.3. 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 lamba’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.)

4.18.4. 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.

`-unreg`:

Compile via vanilla ANSI C only, turning off platform-specific optimisations. NOTE: in order to use `-unreg`, you need to have a set of libraries (including the RTS) built for unregisterised compilation. This amounts to building GHC with way "u" enabled.

4.19. Flag reference

This section is a quick-reference for GHC's command-line flags. For each flag, we also list its static/dynamic status (see Section 4.2), and the flag's opposite (if available).

4.19.1. Help and verbosity options (Section 4.4)

Flag	Description	Static/Dynamic	Reverse
`-?`	help	static	-
`-help`	help	static	-
`-v`	verbose mode (equivalent to `-v3`)	dynamic	-
`-v`*n*	set verbosity level	dynamic	-
`--version`	display GHC version	static	-
`--numeric-version`	display GHC version (numeric only)	static	-
`--print-libdir`	display GHC library directory	static	-

4.19.2. Which phases to run (Section 4.6)

Flag	Description	Static/Dynamic	Reverse
`-E`	Stop after preprocessing (`.hspp` file)	static	-
`-C`	Stop after generating C (`.hc` file)	static	-
`-S`	Stop after generating assembly (`.s` file)	static	-
`-c`	Stop after compiling to object code (`.o` file)	static	-

4.19.3. Redirecting output (Section 4.7)

Flag	Description	Static/Dynamic	Reverse
`-hcsuf` *suffix*	set the suffix to use for intermediate C files	static	-

Flag	Description	Static/Dynamic	Reverse
`-hidir` *dir*	set directory for interface files	static	-
`-hisuf` *suffix*	set the suffix to use for interface files	static	-
`-o` *filename*	set output filename	static	-
`-odir` *dir*	set output directory	static	-
`-ohi` *filename*	set the filename in which to put the interface	static	
`-osuf` *suffix*	set the output file suffix	static	-

4.19.4. Keeping intermediate files (Section 4.7.1)

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

4.19.5. Temporary files (Section 4.7.2)

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

4.19.6. Finding imports (Section 4.9.2)

Flag	Description	Static/Dynamic	Reverse

Flag	Description	Static/Dynamic	Reverse
`-idir1:dir2:...`	add *dir*, *dir2*, etc. to import path	static	-
`-i`	Empty the import directory list	static	-

4.19.7. Interface file options (Section 4.9.4)

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`	Read the interface in *file* and dump it as text to `stdout`.	static	-

4.19.8. Recompilation checking (Section 4.9.5)

Flag	Description	Static/Dynamic	Reverse
`-no-recomp`	Turn off recompilation checking	static	`-recomp`

4.19.9. Interactive-mode options (Section 3.8)

Flag	Description	Static/Dynamic	Reverse
`-ignore-dot-ghci`	Disable reading of `.ghci` files	static	-
`-read-dot-ghci`	Enable reading of `.ghci` files	static	-

4.19.10. Packages (Section 4.10)

Flag	Description	Static/Dynamic	Reverse
`-package` *name*	Use package *name*	static	-
`-package-conf` *file*	Load more packages from *file*	static	-
`-package-name` *name*	Compile code for package *name*	static	-

4.19.11. Language options (Section 7.1)

Flag	Description	Static/Dynamic	Reverse
`-fallow-overlapping-instances`		dynamic	`-fno-allow-overlapping-instances`
`-fallow-undecidable-instances`	Enable undecidable instances	dynamic	`-fno-allow-undecidable-instances`
`-fgenerics`	Enable generics	dynamic	`-fno-fgenerics`
`-fglasgow-exts`	Enable most language extensions	dynamic	`-fno-glasgow-exts`
`-ffi` or `-fffi`	Enable foreign function interface (implied by `-fglasgow-exts`)	dynamic	`-fno-ffi`
`-fwith`	Enable deprecated `with` keyword	dynamic	`-fno-with`
`-fignore-asserts`	Ignore assertions	dynamic	`-fno-ignore-asserts`
`-fno-implicit-prelude`	Don't implicitly `import Prelude`	dynamic	-
`-fno-monomorphism-restriction`	Disable the monomorphism restriction	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-firrefutable-tuples`	Make tuple pattern matching irrefutable	dynamic	`-fno-irrefutable-tuples`
`-fcontext-stack`*n*	set the limit for context reduction	dynamic	-

4.19.12. Warnings (Section 4.8)

Flag	Description	Static/Dynamic	Reverse
`-W`	enable normal warnings	dynamic	`-w`
`-w`	disable all warnings	dynamic	-
`-Wall`	enable all warnings	dynamic	`-w`
`-fwarn-deprecations`	warn about uses of functions & types that are deprecated	dynamic	`-fno-warn-deprecations`
`-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-incomplete-patterns`	warn when a pattern match could fail	dynamic	`-fno-warn-incomplete-patterns`
`-fwarn-misc`	enable miscellaneous warnings	dynamic	`-fno-warn-misc`
`-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`

Flag	Description	Static/Dynamic	Reverse
`-fwarn-name-shadowing`	warn when names are shadowed	dynamic	`-fno-warn-name-shadowing`
`-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-type-defaults`	warn when defaulting happens	dynamic	`-fno-warn-type-defaults`
`-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`

4.19.13. Optimisation levels (Section 4.11)

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

4.19.14. Individual optimisations (Section 4.11.2)

Flag	Description	Static/Dynamic	Reverse
`-fcase-merge`	Enable case-merging	static	`-fno-case-merge`

Flag	Description	Static/Dynamic	Reverse
-fdicts-strict	Make dictionaries strict	dynamic	-fno-dicts-strict
-fdo-eta-reduction	Enable eta-reduction	static	-fno-do-eta-reduction
-fdo-lambda-eta-expansion	Enable lambda eta-reduction	static	-fno-do-lambda-eta-expansion
-fexcess-precision	Enable excess intermediate precision	static	-fno-excess-precision
-ffoldr-build-on	Enable foldr-build optimisation	static	-fno-foldr-build-on
-fignore-asserts	Ignore assertions in the source	static	-fno-ignore-asserts
-fignore-interface-pragmas	Ignore pragmas in interface files	static	-fno-ignore-interface-pragmas
-flet-no-escape	Enable let-no-escape optimisation	static	-fno-let-no-escape
-fliberate-case-threshold	Tweak the liberate-case optimisation (default: 10)	static	-fno-liberate-case-threshold
-fomit-interface-pragmas	Don't generate interface pragmas	static	-fno-omit-interface-pragmas
-fmax-worker-args	If a worker has that many arguments, none will be unpacked anymore (default: 10)	static	-
-fmax-simplifier-iterations	Set the max iterations for the simplifier	static	-
-fno-cpr	Turn off CPR analysis	static	-
-fno-cse	Turn off common sub-expression	static	-
-fno-pre-inlining	Turn off pre-inlining	static	-

Flag	Description	Static/Dynamic	Reverse
-fno-strictness	Turn off strictness analysis	static	-
-fnumbers-strict	Make numbers strict	dynamic	-fno-numbers-strict
-funbox-strict-fields	Flatten strict constructor fields	static	-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-update-in-place	Tweak unfolding settings	static	-fno-unfolding-update-in-place
-funfolding-use-threshold	Tweak unfolding settings	static	-fno-unfolding-use-threshold
-fusagesp	Turn on UsageSP analysis	static	-fno-usagesp

4.19.15. 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
-auto-dicts	Auto-add _scc_s to all dictionaries	static	-no-auto-dicts
-caf-all	Auto-add _scc_s to all CAFs	static	-no-caf-all

Flag	Description	Static/Dynamic	Reverse
`-prof`	Turn on profiling	static	-
`-ticky`	Turn on ticky-ticky profiling	static	-

4.19.16. Parallelism options (Section 4.14)

Flag	Description	Static/Dynamic	Reverse
`-gransim`	Enable GRANSIM	static	-
`-parallel`	Enable Parallel Haskell	static	-
`-smp`	Enable SMP support	static	-

4.19.17. C pre-processor options (Section 4.12.3)

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	static	-

4.19.18. C compiler options (Section 4.12.5)

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

4.19.19. Code generation options (Section 4.12.6)

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	static	-

4.19.20. Linking options (Section 4.12.7)

Flag	Description	Static/Dynamic	Reverse
`-dynamic`	Use dynamic Haskell libraries (if available)	static	-
`-l`*`lib`*	Link in library *`lib`*	static	-
`-L`*`dir`*	Add *`dir`* to the list of directories searched for libraries	static	-
`-package` *`name`*	Link in package *`name`*	static	-
`-framework` *`name`*	On Darwin/MacOS X only, link in the framework *`name`*. This option corresponds to the `-framework` option for Apple's Linker.	static	-
`-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.	static	-
`-split-objs`	Split objects (for libraries)	static	-
`-static`	Use static Haskell libraries	static	-
`-no-hs-main`	Don't asssume this program contains `main`	static	-

4.19.21. Replacing phases (Section 4.12.1)

Flag	Description	Static/Dynamic	Reverse
`-pgmL` *cmd*	Use *cmd* as the literate pre-processor	static	-
`-pgmP` *cmd*	Use *cmd* as the C pre-processor (with `-cpp` only)	static	-
`-pgmc` *cmd*	Use *cmd* as the C compiler	static	-
`-pgma` *cmd*	Use *cmd* as the assembler	static	-
`-pgml` *cmd*	Use *cmd* as the linker	static	-
`-pgmdll` *cmd*	Use *cmd* as the DLL generator	static	-
`-pgmdep` *cmd*	Use *cmd* as the dependency generator	static	-
`-pgmF` *cmd*	Use *cmd* as the pre-processor (with `-F` only)	static	-

4.19.22. Forcing options to particular phases (Section 4.12.2)

Flag	Description	Static/Dynamic	Reverse
`-optL` *option*	pass *option* to the literate pre-processor	dynamic	-
`-optP` *option*	pass *option* to cpp (with `-cpp` only)	dynamic	-
`-optc` *option*	pass *option* to the C compiler	dynamic	-
`-opta` *option*	pass *option* to the assembler	dynamic	-
`-optl` *option*	pass *option* to the linker	static	-
`-optdll` *option*	pass *option* to the DLL generator	static	-
`-optdep` *option*	pass *option* to the dependency generator	static	-

4.19.23. Platform-specific options (Section 4.15)

Flag	Description	Static/Dynamic	Reverse
`-mv8`	(SPARC only) enable version 8 support	static	-
`-monly-[32]-regs`	(x86 only) give some registers back to the C compiler	dynamic	-

4.19.24. External core file options (Section 4.17)

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

4.19.25. Compiler debugging options (Section 4.18)

Flag	Description	Static/Dynamic	Reverse
`-dcore-lint`	Turn on internal sanity checking	dynamic	-
`-ddump-absC`	Dump abstract C	dynamic	-
`-ddump-asm`	Dump assembly	dynamic	-
`-ddump-bcos`	Dump interpreter byte code	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-inlinings`	Dump inlining info	dynamic	-
`-ddump-occur-anal`	Dump occurrence analysis output	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-ddump-parsed`	Dump parse tree	dynamic	-
`-ddump-realC`	Dump “real” C	dynamic	-
`-ddump-rn`	Dump renamer output	dynamic	-
`-ddump-rules`	Dump rules	dynamic	-
`-ddump-sat`	Dump saturated output	dynamic	-
`-ddump-simpl`	Dump final simplifier output	dynamic	-
`-ddump-simpl-iterations`	Dump output from each simplifier iteration	dynamic	-
`-ddump-spec`	Dump specialiser output	dynamic	-
`-ddump-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-usagesp`	Dump UsageSP analysis output	dynamic	-
`-ddump-worker-wrapper`	Dump worker-wrapper output	dynamic	-
`-ddump-rn-trace`	Trace renamer	dynamic	-
`-ddump-rn-stats`	Renamer stats	dynamic	-
`-ddump-stix`	Native code generator intermediate form	dynamic	-
`-ddump-simpl-stats`	Dump simplifier stats	dynamic	-
`-dppr-debug`	Turn on debug printing (more verbose)	static	-
`-dppr-noprags`	Don’t output pragma info in dumps	static	-
`-dppr-user-length`	Set the depth for printing expressions in error msgs	static	-
`-dsource-stats`	Dump haskell source stats	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-dstg-lint`	STG pass sanity checking	dynamic	-
`-dstg-stats`	Dump STG stats	dynamic	-
`-dusagesp-lint`	UsageSP sanity checker	dynamic	-
`-dverbose-core2core`	Show output from each core-to-core pass	dynamic	-
`-dverbose-stg2stg`	Show output from each STG-to-STG pass	dynamic	-
`-unreg`	Enable unregisterised compilation	static	-

4.19.26. Misc compiler options

Flag	Description	Static/Dynamic	Reverse
`-funfold-casms-in-hi-file`	Allow casms in unfoldings	static	-
`-femit-extern-decls`	???	static	-
`-fglobalise-toplev-names`	Make all top-level names global (for `-split-objs`)	static	-
`-fno-hi-version-check`	Don't complain about `.hi` file mismatches	static	-
`-dno-black-holing`	Turn off black holing (probably doesn't work)	static	-
`-fno-method-sharing`	Don't share specialisations of overloaded functions	static	-
`-fno-prune-decls`	Renamer: don't prune declarations	static	-

Flag	Description	Static/Dynamic	Reverse
`-fno-prune-tydecls`	Renamer: don't prune type declarations	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)	static	-

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.
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. The XML profiling tool (described in Section 5.5) will be able to display real loops in the call-graph.

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 aribrary 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.

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`>.

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.)

`-px`:

The `-px` option generates profiling information in the XML format understood by our new profiling tool, see Section 5.5.

`-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 4.16.3.

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 ativity, 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 5.6.
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's one more option which relates 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.

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 A is retained by an object B if object A can be reached by recursively following pointers starting from object B but not meeting any other retainers on the way. Each object has one or more retainers, collectively called its *retainer set*.

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 compuation 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.

5.5. Graphical time/allocation profile

You can view the time and allocation profiling graph of your program graphically, using **ghcprof**. This is a new tool with GHC 4.08, and will eventually be the de-facto standard way of viewing GHC profiles[2]

To run **ghcprof**, you need daVinci installed, which can be obtained from *The Graph Visualisation Tool daVinci* (http://www.informatik.uni-bremen.de/daVinci/). Install one of the binary distributions[3], and set your DAVINCIHOME environment variable to point to the installation directory.

ghcprof uses an XML-based profiling log format, and you therefore need to run your program with a different option: `-px`. The file generated is still called `<prog>.prof`. To see the profile, run **ghcprof** like this:

```
$ ghcprof <prog>.prof
```

which should pop up a window showing the call-graph of your program in glorious detail. More information on using **ghcprof** can be found at *The Cost-Centre Stack Profiling Tool for GHC* (http://www.dcs.warwick.ac.uk/people/academic/Stephen.Jarvis/profiler/index.html).

5.6. 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.

5.7. 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 appropriate libraries and things when you made the system. See "Customising what libraries to build," in the installation guide.

To get your compiled program to spit out the ticky-ticky numbers, use a `-r` RTS option. See Section 4.16.

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!

Notes

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

2. Actually this isn't true any more, we are working on a new tool for displaying heap profiles using Gtk+HS, so **ghcprof** may go away at some point in the future.
3. daVinci is sadly not open-source :-(.

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 -Sstderr -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 would help.

If the heap size is approaching the maximum (64M by default), and you have lots of memory, try increasing the maximum with the `-M<size>` option, e.g.: **ghc -c -O -M1024m Foo.hs**.

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

Deeply-nested list comprehensions seem to be one such; in the past, very large constant tables were bad, too.

We'd rather you reported such behaviour as a bug, so that we can try to correct it.

The part of the compiler that is occasionally prone to wandering off for a long time is the strictness analyser. You can turn this off individually with `-fno-strictness`.

To figure out which part of the compiler is badly behaved, the `-v2` option is your friend.

If your module has big wads of constant data, GHC may produce a huge basic block that will cause the native-code generator's register allocator to founder. Bring on `-fvia-C` (not that GCC will be that quick about it, either).

Explicit `import` declarations:

Instead of saying `import Foo`, say `import Foo (...stuff I want...)` You can get GHC to tell you the minimal set of required imports by using the `-ddump-minimal-imports` option (see Section 4.9.4).

Truthfully, the reduction on compilation time will be very small. However, judicious use of `import` declarations can make a program easier to understand, so it may be a good idea anyway.

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 the HBC library, but it will take you much longer than typing `import QSort`. (Incidentally, it doesn't hurt if the Someone Else is Lennart Augustsson.)

Please report any overly-slow GHC-compiled programs. The current definition of "overly-slow" is "the HBC-compiled version ran faster"...

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.
>
> At the moment, if you turn on `-O` you get GCC instead. This may change in the future.
>
> 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 7.6.3 and Section 7.6.4.

"But how do I know where overloading is creeping in?":

A low-tech way: grep (search) your interface files for overloaded type signatures; e.g.,:

```
% egrep '^[a-z].*::.*=>' *.hi
```

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 7.6.1.

Explicit `export` list:

If you do not have an explicit export list in a module, GHC must assume that everything in that module will be exported. This has various pessimising effects. For example, if a bit of code is actually *unused* (perhaps because of unfolding effects), GHC will not be able to throw it away, because it is exported and some other module may be relying on its existence.

GHC can be quite a bit more aggressive with pieces of code if it knows they are not exported.

Look at the Core syntax!

(The form in which GHC manipulates your code.) Just run your compilation with `-ddump-simpl` (don't forget the `-O`).

If profiling has pointed the finger at particular functions, look at their Core code. `lets` are bad, `cases` are good, dictionaries (`d.<Class>.<Unique>`) [or anything overloading-ish] are bad, nested lambdas are bad, explicit data constructors are good, primitive operations (e.g., `eqInt#`) are good,. . .

Use 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.

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 `Float`s:

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 `Float`s. They take up half the space in the heap compared to `Doubles`. However, this isn't true on a 64-bit machine.

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 4.16.2).

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 -Sstderr`, 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 -Sstderr -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. To use them, you'll need to give a `-fglasgow-exts` option.

Virtually all 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-standard 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

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

`-fglasgow-exts`:

This simultaneously enables all of the extensions to Haskell 98 described in Chapter 7, except where otherwise noted.

`-ffi` and `-fffi`:

This option enables the language extension defined in the Haskell 98 Foreign Function Interface Addendum plus deprecated syntax of previous versions of the FFI for backwards compatibility.

`-fwith`:

This option enables the deprecated `with` keyword for implicit parameters; it is merely provided for backwards compatibility. It is independent of the `-fglasgow-exts` flag.

`-fno-monomorphism-restriction`:

Switch off the Haskell 98 monomorphism restriction. Independent of the `-fglasgow-exts` flag.

`-fallow-overlapping-instances`
`-fallow-undecidable-instances`
`-fallow-incoherent-instances`
`-fcontext-stack`

See Section 7.3.5.3. Only relevant if you also use `-fglasgow-exts`.

`-finline-phase`

See Section 7.7. Only relevant if you also use `-fglasgow-exts`.

`-fgenerics`

See Section 7.8. Independent of `-fglasgow-exts`.

`-fno-implicit-prelude`

GHC normally imports `Prelude.hi` files for you. If you'd rather it didn't, then give it a `-fno-implicit-prelude` 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.)

Even though you have not imported the Prelude, most of the built-in syntax still refers to the built-in Haskell Prelude types and values, as specified by the Haskell Report. For example, the type `[Int]` still means `Prelude.[] Int`; tuples continue to refer to the standard Prelude tuples; the translation for list comprehensions continues to use `Prelude.map` etc.

However, `-fno-implicit-prelude` does change the handling of certain built-in syntax: see Section 7.5.4.

7.2. Unboxed types and primitive operations

GHC is built on a raft of primitive data types and operations. 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.

We do not currently have good, up-to-date documentation about the primitives, perhaps because they are mainly intended for internal use. There used to be a long section about them here in the User Guide, but it became out of date, and wrong information is worse than none.

The Real Truth about what primitive types there are, and what operations work over those types, is held in the file `fptools/ghc/compiler/prelude/primops.txt`. This file is used directly to generate GHC's primitive-operation definitions, so it is always correct! It is also intended for processing into text.

Indeed, the result of such processing is part of the description of the External Core language (http://haskell.cs.yale.edu/ghc/docs/papers/core.ps.gz). So that document is a good place to look for a type-set version. We would be very happy if someone wanted to volunteer to produce an SGML back end to the program that processes `primops.txt` so that we could include the results here in the User Guide.

What follows here is a brief summary of some main points.

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 that primitive types, values, and operations have a `#` suffix.

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.

There are some restrictions on the use of primitive types, the main one being 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).

Nevertheless, A numerically-intensive program using unboxed types can go a *lot* faster than its "standard" counterpart—we saw a threefold speedup on one example.

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 this section return unboxed tuples.

There are some pretty stringent restrictions on the use of unboxed tuples:

- 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.
- Unboxed tuples may only be constructed as the direct result of a function, and may only be deconstructed with a `case` expression. eg. the following are valid:

  ```
  f x y = (# x+1, y-1 #)
  g x = case f x x of { (# a, b #) -> a + b }
  ```

  but the following are invalid:

  ```
  f x y = g (# x, y #)
  g (# x, y #) = x + y
  ```

- No variable can have an unboxed tuple type. This is illegal:

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

  because `x` has an unboxed tuple type.

Note: we may relax some of these restrictions in the future.

The `IO` and `ST` monads use unboxed tuples to avoid unnecessary allocation during sequences of operations.

7.3. Type system extensions

7.3.1. 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 `*` then an explicit kind annotation must be used (see Section 7.3.3).

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

7.3.2. Infix type constructors

GHC allows type constructors to be operators, and to be written infix, very much like expressions. More specifically:

- A type constructor can be an operator, beginning with a colon; e.g. `:*:`. The lexical syntax is the same as that for data constructors.
- Types can be written infix. For example `Int :*: Bool`.
- 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 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.)
- Data type and type-synonym declarations can be written infix. E.g.

  ```
  data a :*: b = Foo a b
  type a :+: b = Either a b
  ```

- The only thing that differs between operators in types and operators in expressions is that ordinary non-constructor operators, such as `+` and `*` are not allowed in types. Reason: the uniform thing to do would be to make them type variables, but that's not very useful. A less uniform but more useful thing would be to allow them to be type *constructors*. But that gives trouble in export lists. So for now we just exclude them.

7.3.3. 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. Namely:

- `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.

7.3.4. 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`).

With the `-fglasgow-exts` GHC lifts this restriction.

7.3.5. Multi-parameter type classes

This section documents GHC's implementation of multi-parameter type classes. There's lots of background in the paper Type classes: exploring the design space (http://research.microsoft.com/~simonpj/multi.ps.gz) (Simon Peyton Jones, Mark Jones, Erik Meijer).

I'd like to thank people who reported shorcomings in the GHC 3.02 implementation. Our default decisions were all conservative ones, and the experience of these heroic pioneers has given useful concrete examples to support several generalisations. (These appear below as design choices not implemented in 3.02.)

I've discussed these notes with Mark Jones, and I believe that Hugs will migrate towards the same design choices as I outline here. Thanks to him, and to many others who have offered very useful feedback.

7.3.5.1. Types

There are the following restrictions on the form of a qualified type:

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

(Here, I write the "foralls" explicitly, although the Haskell source language omits them; in Haskell 1.4, 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 7.3.9).

1. *Each universally quantified type variable `tvi` must be mentioned (i.e. appear free) in `type`.* The reason for this is that a value with a type that does not obey this restriction could not be used without introducing ambiguity. 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`.

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.

These restrictions apply to all types, whether declared in a type signature or inferred.

Unlike Haskell 1.4, constraints in types do *not* have to be of the form *(class type-variables)*. Thus, these type signatures are perfectly OK

```
f :: Eq (m a) => [m a] -> [m a]
g :: Eq [a] => ...
```

This choice recovers principal types, a property that Haskell 1.4 does not have.

7.3.5.2. Class declarations

1. *Multi-parameter type classes are permitted*. For example:

   ```
   class Collection c a where
     union :: c a -> c a -> c a
     ...etc.
   ```

2. *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`.)

3. *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
```

4. *In the signature of a class operation, every constraint must mention at least one type variable that is not a class type variable*. Thus:

```
class Collection c a where
  mapC :: Collection c b => (a->b) -> c a -> c b
```

is OK because the constraint `(Collection a b)` mentions `b`, even though it also mentions the class variable `a`. On the other hand:

```
class C a where
  op :: Eq a => (a,b) -> (a,b)
```

is not OK because the constraint `(Eq a)` mentions on the class type variable `a`, but not `b`. However, any such example is easily fixed by moving the offending context up to the superclass context:

```
class Eq a => C a where
  op ::(a,b) -> (a,b)
```

A yet more relaxed rule would allow the context of a class-op signature to mention only class type variables. However, that conflicts with Rule 1(b) for types above.

5. *The type of each class operation must mention all of the class type variables*. 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`. This rule is a consequence of Rule 1(a), above, for types, and has the same motivation. Sometimes, offending class declarations exhibit misunderstandings. For example, `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
```

7.3.5.3. Instance declarations

1. *Instance declarations may not overlap*. The two instance declarations

   ```
   instance context1 => C type1 where ...
   instance context2 => C type2 where ...
   ```

   "overlap" if `type1` and `type2` unify However, if you give the command line option `-fallow-overlapping-instances` then overlapping instance declarations are permitted. However, GHC arranges never to commit to using an instance declaration if another instance declaration also applies, either now or later.

   - EITHER `type1` and `type2` do not unify
   - OR `type2` is a substitution instance of `type1` (but not identical to `type1`), or vice versa.

   Notice that these rules

   - make it clear which instance decl to use (pick the most specific one that matches)
   - do not mention the contexts `context1`, `context2` Reason: you can pick which instance decl "matches" based on the type.

   However the rules are over-conservative. Two instance declarations can overlap, but it can still be clear in particular situations which to use. For example:

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

   These are rejected by GHC's rules, but it is clear what to do when trying to solve the constraint `C (Int,Int)` because the second instance cannot apply. Yell if this restriction bites you.

   GHC is also conservative about committing to an overlapping instance. For example:

   ```
   class C a where { op :: a -> a }
   instance C [Int] where ...
   instance C a => C [a] where ...

   f :: C b => [b] -> [b]
   f x = op x
   ```

   From the RHS of f we get the constraint `C [b]`. But GHC does not commit to the second instance declaration, because in a paricular call of f, b might be instantiate to Int, so the first instance declaration would be appropriate. So GHC rejects the program. If you add

`-fallow-incoherent-instances` GHC will instead silently pick the second instance, without complaining about the problem of subsequent instantiations.

Regrettably, GHC doesn't guarantee to detect overlapping instance declarations if they appear in different modules. GHC can "see" the instance declarations in the transitive closure of all the modules imported by the one being compiled, so it can "see" all instance decls when it is compiling `Main`. However, it currently chooses not to look at ones that can't possibly be of use in the module currently being compiled, in the interests of efficiency. (Perhaps we should change that decision, at least for `Main`.)

2. *There are no restrictions on the type in an instance head, except that at least one must not be a type variable*. The instance "head" is the bit after the "=>" in an instance decl. For example, these are OK:

```
instance C Int a where ...

instance D (Int, Int) where ...

instance E [[a]] where ...
```

Note that instance heads *may* contain repeated type variables. For example, this is OK:

```
instance Stateful (ST s) (MutVar s) where ...
```

The "at least one not a type variable" restriction is to ensure that context reduction terminates: each reduction step removes one type constructor. For example, the following would make the type checker loop if it wasn't excluded:

```
instance C a => C a where ...
```

There are two situations in which the rule is a bit of a pain. First, 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
```

Second, 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) => ...
```

I'm on the lookout for a simple rule that preserves decidability while allowing these idioms. The experimental flag `-fallow-undecidable-instances` lifts this restriction, allowing all the types in an instance head to be type variables.

3. *Unlike Haskell 1.4, instance heads may use type synonyms*. As always, using a type synonym is just shorthand for writing the RHS of the type synonym definition. For example:

   ```
   type Point = (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.

4. *The types in an instance-declaration context must all be type variables*. Thus

   ```
   instance C a b => Eq (a,b) where ...
   ```

   is OK, but

   ```
   instance C Int b => Foo b where ...
   ```

   is not OK. Again, the intent here is to make sure that context reduction terminates. Voluminous correspondence on the Haskell mailing list has convinced me that it's worth experimenting with a more liberal rule. If you use the flag `-fallow-undecidable-instances` can use arbitrary types in an instance context. 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*.

7.3.6. Implicit parameters

Implicit paramters 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, stil rather incomplete, documentation is due to Jeff Lewis.)

A variable is called *dynamically bound* when it is bound by the calling context of a function and *statically bound* when bound by the callee's context. In Haskell, all variables are statically bound. Dynamic binding of variables is a notion that goes back to Lisp, but was later discarded in more modern incarnations, such as Scheme. Dynamic binding can be very confusing in an untyped language, and unfortunately, typed languages, in particular Hindley-Milner typed languages like Haskell, only support static scoping of variables.

However, by a simple extension to the type class system of Haskell, we can support dynamic binding. Basically, we express the use of a dynamically bound variable as a constraint on the type. These constraints lead to types of the form `(?x::t') => t`, which says "this function uses a dynamically-bound variable `?x` of type `t'`". For example, the following expresses the type of a sort function, implicitly 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 is introduced by the special form `?x`, where `x` is any valid identifier. Use if this construct also introduces new dynamic binding constraints. 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
```

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 = fst (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 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.

An implicit parameter is bound using the standard `let` binding form, where the bindings 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. This form binds the implicit parameters arising in the body, not the free variables as a `let` or `where` would do. For example, we define the `min` function by binding `cmp`.

```
min :: [a] -> a
min  = let ?cmp = (<=) in least
```

Note the following additional 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.

7.3.7. Linear implicit parameters

Linear implicit parameters are an idea developed by Koen Claessen, Mark Shields, and Simon PJ. They address the long-standing problem that monads seem over-kill for certain sorts of problem, notably:

- distributing a supply of unique names
- distributing a suppply of random numbers
- distributing an oracle (as in QuickCheck)

Linear implicit parameters are just like ordinary implicit parameters, except that they are "linear" -- that is, they cannot be copied, and must be explicitly "split" instead. Linear implicit parameters are written '`%x`' instead of '`?x`'. (The '/' in the '%' suggests the split!)

For example:

```
    import GHC.Exts( Splittable )

    data NameSupply = ...

    splitNS :: NameSupply -> (NameSupply, NameSupply)
    newName :: NameSupply -> Name

    instance Splittable NameSupply where
 split = splitNS


    f :: (%ns :: NameSupply) => Env -> Expr -> Expr
    f env (Lam x e) = Lam x' (f env e)
      where
        x'   = newName %ns
        env' = extend env x x'
    ...more equations for f...
```

Notice that the implicit parameter %ns is consumed

- once by the call to `newName`
- once by the recursive call to `f`

So the translation done by the type checker makes the parameter explicit:

```
f :: NameSupply -> Env -> Expr -> Expr
f ns env (Lam x e) = Lam x' (f ns1 env e)
     where
  (ns1,ns2) = splitNS ns
x' = newName ns2
env = extend env x x'
```

Notice the call to 'split' introduced by the type checker. How did it know to use 'splitNS'? Because what it really did was to introduce a call to the overloaded function 'split', defined by the class `Splittable`:

```
class Splittable a where
  split :: a -> (a,a)
```

The instance for `Splittable NameSupply` tells GHC how to implement split for name supplies. But we can simply write

```
g x = (x, %ns, %ns)
```

and GHC will infer

```
g :: (Splittable a, %ns :: a) => b -> (b,a,a)
```

The `Splittable` class is built into GHC. It's exported by module `GHC.Exts`.

Other points:

- '`?x`' and '`%x`' are entirely distinct implicit parameters: you can use them together and they won't intefere with each other.
- You can bind linear implicit parameters in 'with' clauses.
- You cannot have implicit parameters (whether linear or not) in the context of a class or instance declaration.

7.3.7.1. Warnings

The monomorphism restriction is even more important than usual. Consider the example above:

```
f :: (%ns :: NameSupply) => Env -> Expr -> Expr
f env (Lam x e) = Lam x' (f env e)
  where
```

```
x'   = newName %ns
env' = extend env x x'
```

If we replaced the two occurrences of x' by (newName %ns), which is usually a harmless thing to do, we get:

```
f :: (%ns :: NameSupply) => Env -> Expr -> Expr
f env (Lam x e) = Lam (newName %ns) (f env e)
  where
    env' = extend env x (newName %ns)
```

But now the name supply is consumed in *three* places (the two calls to newName,and the recursive call to f), so the result is utterly different. Urk! We don't even have the beta rule.

Well, this is an experimental change. With implicit parameters we have already lost beta reduction anyway, and (as John Launchbury puts it) we can't sensibly reason about Haskell programs without knowing their typing.

7.3.7.2. Recursive functions

Linear implicit parameters can be particularly tricky when you have a recursive function Consider

```
foo :: %x::T => Int -> [Int]
foo 0 = []
foo n = %x : foo (n-1)
```

where T is some type in class Splittable.

Do you get a list of all the same T's or all different T's (assuming that split gives two distinct T's back)?

If you supply the type signature, taking advantage of polymorphic recursion, you get what you'd probably expect. Here's the translated term, where the implicit param is made explicit:

```
foo x 0 = []
foo x n = let (x1,x2) = split x
          in x1 : foo x2 (n-1)
```

But if you don't supply a type signature, GHC uses the Hindley Milner trick of using a single monomorphic instance of the function for the recursive calls. That is what makes Hindley Milner type inference work. So the translation becomes

```
foo x = let
          foom 0 = []
          foom n = x : foom (n-1)
        in
        foom
```

Result: 'x' is not split, and you get a list of identical T's. So the semantics of the program depends on whether or not foo has a type signature. Yikes!

You may say that this is a good reason to dislike linear implicit parameters and you'd be right. That is why they are an experimental feature.

7.3.8. Functional dependencies

Functional dependencies are implemented as described by Mark Jones in "Type Classes with Functional Dependencies (http://www.cse.ogi.edu/~mpj/pubs/fundeps.html)", Mark P. Jones, In Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany, March 2000, Springer-Verlag LNCS 1782, .

There should be more documentation, but there isn't (yet). Yell if you need it.

7.3.9. 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
```

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 arrrow. As `g2` shows, the polymorphic type on the left of the function arrow can be overloaded.

The functions `f3` and `g3` have rank-3 types; they have rank-2 types on the left of a function arrow.

GHC allows types of arbitrary rank; you can nest foralls arbitrarily deep in function arrows. (GHC used to be restricted to rank 2, but that restriction has now been lifted.) In particular, a forall-type (also called a "type scheme"), including an operational type class context, is legal:

- On the left of a function arrow
- On the right of a function arrow (see Section 7.3.11)
- 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,g3` above would be valid field type signatures.
- As the type of an implicit parameter
- In a pattern type signature (see Section 7.3.13)

There is one place you cannot put a `forall`: you cannot instantiate a type variable with a forall-type. So you cannot make a forall-type the argument of a type constructor. So these types are illegal:

```
x1 :: [forall a. a->a]
x2 :: (forall a. a->a, Int)
x3 :: Maybe (forall a. a->a)
```

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

7.3.9.1. 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.

7.3.9.2. Type inference

In general, type inference for arbitrary-rank types is undecideable. 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 7.3.13), 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.

7.3.9.3. 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.

7.3.10. Liberalised type synonyms

Type synonmys are like macros at the type level, and 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,Bool)    -- A rank-2 type
  g f = f Int True
  ```

- 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 epxanding 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.

7.3.11. For-all hoisting

It is often convenient to use generalised type synonyms at the right hand end of an arrow, thus:

```
type Discard a = forall b. a -> b -> a

g :: Int -> Discard Int
g x y z = x+y
```

Simply expanding the type synonym would give

```
g :: Int -> (forall b. Int -> b -> Int)
```

but GHC "hoists" the `forall` to give the isomorphic type

```
g :: forall b. Int -> Int -> b -> Int
```

In general, the rule is this: *to determine the type specified by any explicit user-written type (e.g. in a type signature), GHC expands type synonyms and then repeatedly performs the transformation:*

```
  type1 -> forall a1..an. context2 => type2
==>
  forall a1..an. context2 => type1 -> type2
```

(In fact, GHC tries to retain as much synonym information as possible for use in error messages, but that is a usability issue.) This rule applies, of course, whether or not the `forall` comes from a synonym. For example, here is another valid way to write `g`'s type signature:

```
g :: Int -> Int -> forall b. b -> Int
```

When doing this hoisting operation, GHC eliminates duplicate constraints. For example:

```
type Foo a = (?x::Int) => Bool -> a
g :: Foo (Foo Int)
```

means

```
g :: (?x::Int) => Bool -> Bool -> Int
```

7.3.12. Existentially quantified data constructors

The idea of using existential quantification in data type declarations was suggested by Laufer (I believe, thought doubtless someone will correct me), and implemented in Hope+. 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 heterogenous 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.

7.3.12.1. 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.

7.3.12.2. Type classes

An easy extension (implemented in **hbc**) 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.

Notice the way that the syntax fits smoothly with that used for universal quantification earlier.

7.3.12.3. 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` 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!

7.3.13. Scoped type variables

A *pattern type signature* can introduce a *scoped type variable*. For example

```
f (xs::[a]) = ys ++ ys
          where
             ys :: [a]
             ys = reverse xs
```

The pattern `(xs::[a])` includes a type signature for `xs`. This brings the type variable `a` into scope; it scopes over all the patterns and right hand sides for this equation for `f`. In particular, it is in scope at the type signature for `y`.

Pattern type signatures are completely orthogonal to ordinary, separate type signatures. The two can be used independently or together. At ordinary type signatures, such as that for `ys`, any type variables mentioned in the type signature *that are not in scope* are implicitly universally quantified. (If there are no type variables in scope, all type variables mentioned in the signature are universally quantified, which is just as in Haskell 98.) In this case, since `a` is in scope, it is not universally quantified, so the type of `ys` is the same as that of `xs`. 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.

Scoped type variables are implemented in both GHC and Hugs. Where the implementations differ from the specification below, those differences are noted.

So much for the basic idea. Here are the details.

7.3.13.1. What a pattern type signature means

A type variable brought into scope by a pattern type signature is simply the name for a type. The restriction they express is that all occurrences of the same name mean the same type. For example:

```
f :: [Int] -> Int -> Int
f (xs::[a]) (y::a) = (head xs + y) :: a
```

The pattern type signatures on the left hand side of `f` express the fact that `xs` must be a list of things of some type `a`; and that `y` must have this same type. The type signature on the expression `(head xs)` specifies that this expression must have the same type `a`. *There is no requirement that the type named by "`a`" is in fact a type variable*. Indeed, in this case, the type named by "`a`" is `Int`. (This is a slight liberalisation from the original rather complex rules, which specified that a pattern-bound type variable should be universally quantified.) For example, all of these are legal:

```
t (x::a) (y::a) = x+y*2

f (x::a) (y::b) = [x,y]       -- a unifies with b

g (x::a) = x + 1::Int         -- a unifies with Int

h x = let k (y::a) = [x,y]    -- a is free in the
      in k x                  -- environment

k (x::a) True    = ...        -- a unifies with Int
k (x::Int) False = ...

w :: [b] -> [b]
w (x::a) = x                  -- a unifies with [b]
```

7.3.13.2. Scope and implicit quantification

- All the type variables mentioned in a pattern, that are not already in scope, are brought into scope by the pattern. We describe this set as the *type variables bound by the pattern*. For example:

  ```
  f (x::a) = let g (y::(a,b)) = fst y
             in
             g (x,True)
  ```

  The pattern `(x::a)` brings the type variable `a` into scope, as well as the term variable `x`. The pattern `(y::(a,b))` contains an occurrence of the already-in-scope type variable `a`, and brings into scope the type variable `b`.

- The type variable(s) bound by the pattern have the same scope as the term variable(s) bound by the pattern. For example:

  ```
  let
    f (x::a) = <...rhs of f...>
  ```

```
    (p::b, q::b) = (1,2)
  in <...body of let...>
```

Here, the type variable `a` scopes over the right hand side of `f`, just like `x` does; while the type variable `b` scopes over the body of the `let`, and all the other definitions in the `let`, just like `p` and `q` do. Indeed, the newly bound type variables also scope over any ordinary, separate type signatures in the `let` group.

- The type variables bound by the pattern may be mentioned in ordinary type signatures or pattern type signatures anywhere within their scope.
- In ordinary type signatures, any type variable mentioned in the signature that is in scope is *not* universally quantified.
- Ordinary type signatures do not bring any new type variables into scope (except in the type signature itself!). So this is illegal:

  ```
  f :: a -> a
  f x = x::a
  ```

  It's illegal because `a` is not in scope in the body of `f`, so the ordinary signature `x::a` is equivalent to `x::forall a.a`; and that is an incorrect typing.
- The pattern type signature is a monotype:
  - A pattern type signature cannot contain any explicit `forall` quantification.
  - The type variables bound by a pattern type signature can only be instantiated to monotypes, not to type schemes.
  - There is no implicit universal quantification on pattern type signatures (in contrast to ordinary type signatures).

- 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
  ```

  (Not implemented in Hugs yet, Dec 98).

7.3.13.3. Result type signatures

- The result type of a function can be given a signature, thus:

  ```
  f (x::a) :: [a] = [x,x,x]
  ```

  The final `:: [a]` after all the patterns gives a signature to the result type. Sometimes this is the only way of naming the type variable you want:

  ```
  f :: Int -> [a] -> [a]
  f n :: ([a] -> [a]) = let g (x::a, y::a) = (y,x)
                        in \xs -> map g (reverse xs `zip` xs)
  ```

Result type signatures are not yet implemented in Hugs.

7.3.13.4. Where a pattern type signature can occur

A pattern type signature can occur in any pattern. For example:

- A pattern type signature can be on an arbitrary sub-pattern, not ust on a variable:

  ```
  f ((x,y)::(a,b)) = (y,x) :: (b,a)
  ```

- Pattern type signatures, including the result part, can be used in lambda abstractions:

  ```
  (\ (x::a, y) :: a -> x)
  ```

- Pattern type signatures, including the result part, can be used in `case` expressions:

  ```
  case e of { (x::a, y) :: a -> x }
  ```

- To avoid ambiguity, the type after the "`::`" in a result pattern signature on a lambda or `case` must be atomic (i.e. a single token or a parenthesised type of some sort). To see why, consider how one would parse this:

  ```
  \ x :: a -> b -> x
  ```

- Pattern type signatures can bind existential type variables. For example:

  ```
  data T = forall a. MkT [a]

  f :: T -> T
  f (MkT [t::a]) = MkT t3
                 where
                   t3::[a] = [t,t,t]
  ```

- Pattern type signatures can be used in pattern bindings:

  ```
  f x = let (y, z::a) = x in ...
  ```

```
f1 x                = let (y, z::Int) = x in ...
f2 (x::(Int,a))     = let (y, z::a)   = x in ...
f3 :: (b->b)        = \x -> x
```

In all such cases, the binding is not generalised over the pattern-bound type variables. Thus `f3` is monomorphic; `f3` has type `b -> b` for some type `b`, and *not* `forall b. b -> b`. In contrast, the binding

```
f4 :: b->b
f4 = \x -> x
```

makes a polymorphic function, but `b` is not in scope anywhere in `f4`'s scope.

7.4. Assertions

If you want to make use of assertions in your standard Haskell code, you could define a function like the following:

```
assert :: Bool -> a -> a
assert False x = error "assertion failed!"
assert _     x = x
```

which works, but gives you back a less than useful error message -- an assertion failed, but which and where?

One way out is to define an extended `assert` function which also takes a descriptive string to include in the error message and perhaps combine this with the use of a pre-processor which inserts the source location where `assert` was used.

Ghc offers a helping hand here, doing all of this for you. For every use of `assert` in the user's source:

```
kelvinToC :: Double -> Double
kelvinToC k = assert (k >= 0.0) (k+273.15)
```

Ghc will rewrite this to also include the source location where the assertion was made,

```
assert pred val ==> assertError "Main.hs|15" pred val
```

The rewrite is only performed by the compiler when it spots applications of `Control.Exception.assert`, so you can still define and use your own versions of `assert`, should you so wish. If not, import `Control.Exception` to make use `assert` in your code.

To have the compiler ignore uses of assert, use the compiler option `-fignore-asserts`. That is, expressions of the form `assert pred e` will be rewritten to `e`.

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

7.5. Syntactic extensions

7.5.1. 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
```

Hierarchical modules have an impact on the way that GHC searches for files. For a description, see Section 4.9.3.

GHC comes with a large collection of libraries arranged hierarchically; see the accompanying library documentation. There is an ongoing project to create and maintain a stable set of "core" libraries used by several Haskell compilers, and the libraries that GHC comes with represent the current status of that project. For more details, see Haskell Libraries (http://www.haskell.org/~simonmar/libraries/libraries.html).

7.5.2. Pattern guards

The discussion that follows is an abbreviated version of Simon Peyton Jones's original proposal (http://research.microsoft.com/~simonpj/Haskell/guards.html). (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 var1 = case lookup env var1 of
  Nothing -> fail
  Just val1 -> case lookup env var2 of
    Nothing -> fail
    Just val2 -> val1 + val2
where
  fail = val1 + val2
```

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

The semantics should be clear enough. The qualifers 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.

7.5.3. 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.

7.5.4. Rebindable syntax

GHC allows most kinds of built-in syntax to be rebound by the user, to facilitate replacing the `Prelude` with a home-grown version, for example.

You may want 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 `-fno-implicit-prelude` flag causes the following pieces of built-in syntax to refer to *whatever is in scope*, not the Prelude versions:

- Integer and fractional literals mean "`fromInteger 1`" and "`fromRational 3.2`", not the Prelude-qualified versions; both in expressions and in patterns.

  However, the standard Prelude `Eq` class is still used for the equality test necessary for literal patterns.

- Negation (e.g. "`- (f x)`") means "`negate (f x)`" (not `Prelude.negate`).
- In an n+k pattern, the standard Prelude `Ord` class is still used for comparison, but the necessary subtraction uses whatever "`(-)`" is in scope (not "`Prelude.(-)`").
- "Do" notation is translated using whatever functions `(>>=)`, `(>>)`, `fail`, and `return`, are in scope (not the Prelude versions). List comprehensions, and parallel array comprehensions, are unaffected.

Be warned: this is an experimental facility, with fewer checks than usual. In particular, it is essential that the functions GHC finds in scope must have the appropriate types, namely:

```
      fromInteger  :: forall a. (...) => Integer  -> a
fromRational :: forall a. (...) => Rational -> a
negate       :: forall a. (...) => a -> a
(-)          :: forall a. (...) => a -> a -> a
(>>=)       :: forall m a. (...) => m a -> (a -> m b) -> m b
(>>)       :: forall m a. (...) => m a -> m b -> m b
```

```
return      :: forall m a. (...) => a      -> m a
fail        :: forall m a. (...) => String -> m a
```

(The (...) part can be any context including the empty context; that part is up to you.) If the functions don't have the right type, very peculiar things may happen. Use `-dcore-lint` to typecheck the desugared program. If Core Lint is happy you should be all right.

7.6. 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 (silently) ignored.

7.6.1. INLINE pragma

GHC (with `-O`, as always) tries to inline (or "unfold") functions/values that are "small enough," thus avoiding the call overhead and possibly exposing other more-wonderful optimisations.

You will probably see these unfoldings (in Core syntax) in your interface files.

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)

#ifdef __GLASGOW_HASKELL__
{-# INLINE key_function #-}
#endif
```

(You don't need to do the C pre-processor carry-on unless you're going to stick the code through HBC—it doesn't like `INLINE` pragmas.)

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.

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:

```
#ifdef __GLASGOW_HASKELL__
{-# INLINE thenUs #-}
{-# INLINE returnUs #-}
#endif
```

7.6.2. 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` (`NOTINLINE` 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).

7.6.3. 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.

To get very fancy, you can also specify a named function to use for the specialised value, as in:

```
{-# RULES "hammeredLookup" hammeredLookup = blah #-}
```

where `blah` is an implementation of `hammerdLookup` written specialy for `Widget` lookups. It's *Your Responsibility* to make sure that `blah` really behaves as a specialised version of `hammeredLookup`!!!

Note we use the `RULE` pragma here to indicate that `hammeredLookup` applied at a certain type should be replaced by `blah`. See Section 7.6.6 for more information on `RULES`.

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.

7.6.4. 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.

7.6.5. 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.

7.6.6. RULES pragma

The RULES pragma lets you specify rewrite rules. It is described in Section 7.7.

7.6.7. DEPRECATED pragma

The DEPRECATED pragma lets you specify that a particular function, class, or type, is deprecated. There are two forms.

- You can deprecate an entire module thus:

  ```
  module Wibble {-# DEPRECATED "Use Wobble instead" #-} where
    ...
  ```

When you compile any module that import `Wibble`, GHC will print the specified message.

- You can deprecate a function, class, or type, with the following top-level declaration:

  ```
  {-# DEPRECATED f, C, T "Don't use these" #-}
  ```

  When you compile any module that imports and uses any of the specifed entities, GHC will print the specified message.

You can suppress the warnings with the flag `-fno-warn-deprecations`.

7.7. Rewrite rules

The programmer can specify rewrite rules as part of the source program (in a pragma). GHC applies these rewrite rules wherever it can.

Here is an example:

```
{-# RULES
      "map/map"       forall f g xs. map f (map g xs) = map (f.g) xs
#-}
```

7.7.1. Syntax

From a syntactic point of view:

- 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.
- There may be zero or more rules in a `RULES` pragma.
- Layout applies in a `RULES` pragma. Currently no new indentation level is set, so you must lay out your rules starting in the same column as the enclosing definitions.
- 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.
- Rules are automatically exported from a module, just as instance declarations are.

7.7.2. Semantics

From a semantic point of view:

- Rules are only applied if you use the `-O` flag.
- 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.
- The LHS and RHS of a rule are typechecked, and must have the same type.
- GHC makes absolutely no attempt to verify that the LHS and RHS of a rule have the same meaning. That is undecideable 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 epression 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 7.7.4 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.

- In the earlier phases of compilation, GHC inlines *nothing that appears on the LHS of a rule*, because once you have substituted for something you can't match against it (given the simple minded matching). So if you write the rule

  ```
          "map/map"       forall f,g.  map f . map g = map (f.g)
  ```

  this *won't* match the expression `map f (map g xs)`. It will only match something written with explicit use of ".". Well, not quite. It *will* match the expression

  ```
  wibble f g xs
  ```

  where `wibble` is defined:

  ```
  wibble f g = map f . map g
  ```

  because `wibble` will be inlined (it's small). Later on in compilation, GHC starts inlining even things on the LHS of rules, but still leaves the rules enabled. This inlining policy is controlled by the per-simplification-pass flag `-finline-phase`*n*.

- 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.

7.7.3. 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`
- `filter`
- `iterate`, `repeat`
- `zip`, `zipWith`

The following are good consumers:

- List comprehensions
- `array` (on its second argument)
- `length`
- `++` (on its first argument)
- `foldr`
- `map`
- `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.

7.7.4. Specialisation

Rewrite rules can be used to get the same effect as a feature present in earlier version of GHC:

```
{-# SPECIALIZE fromIntegral :: Int8 -> Int16 = int8ToInt16 #-}
```

This told GHC to use `int8ToInt16` instead of `fromIntegral` whenever the latter was called with type `Int8 -> Int16`. That is, rather than specialising the original definition of `fromIntegral` the programmer is promising that it is safe to use `int8ToInt16` instead.

This feature is no longer in GHC. But rewrite rules let you do the same thing:

```
{-# RULES
  "fromIntegral/Int8/Int16" fromIntegral = int8ToInt16
#-}
```

This slightly odd-looking rule instructs GHC to replace `fromIntegral` by `int8ToInt16` *whenever the types match*. Speaking more operationally, GHC adds the type and dictionary applications to get the typed rule

```
forall (d1::Integral Int8) (d2::Num Int16) .
        fromIntegral Int8 Int16 d1 d2 = int8ToInt16
```

What is more, this rule does not need to be in the same file as fromIntegral, unlike the `SPECIALISE` pragmas which currently do (so that they have an original definition available to specialise).

7.7.5. 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 defintion of (say) `build` in `PrelBase.lhs` looks llike 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 `ghc/lib/std/PrelBase.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 `PrelList.lhs`.

7.8. Generic classes

(Note: support for generic classes is currently broken in GHC 5.02).

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" clasuse. Of course, you can put in the where clause and over-ride whichever methods you please.

7.8.1. Using generics

To use generics you need to

- Use the flags `-fglasgow-exts` (to enable the extra syntax), `-fgenerics` (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.)

7.8.2. 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).

7.8.3. 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 consructor 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 havn'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.

7.8.4. 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
```

7.9. 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!

7.9.1. Generalising the deriving clause

GHC now permits such instances to be derived instead, 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.

7.9.2. 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' (S t1...tk vk+1...vn) deriving (c1...cm)
```

where `S` is a type constructor, `t1...tk` are types, `vk+1...vn` are type variables which do not occur in any of the `ti`, and the `ci` are partial applications of classes of the form `C t1'...tj'`. The derived instance declarations are, for each `ci`,

```
instance ci (S t1...tk vk+1...v) => ci (T v1...vp)
```

where `p` is chosen so that `T v1...vp` is of the right *kind* for the last parameter of class `Ci`.

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.

7.10. Concurrent and Parallel Haskell

Concurrent and Parallel Haskell are Glasgow extensions to Haskell which let you structure your program as a group of independent 'threads'.

Concurrent and Parallel Haskell have very different purposes.

Concurrent Haskell is for applications which have an inherent structure of interacting, concurrent tasks (i.e. 'threads'). Threads in such programs may be *required*. For example, if a concurrent thread has been spawned to handle a mouse click, it isn't optional—the user wants something done!

A Concurrent Haskell program implies multiple 'threads' running within a single Unix process on a single processor.

You will find at least one paper about Concurrent Haskell hanging off of Simon Peyton Jones's Web page (http://research.microsoft.com/~simonpj/).

Parallel Haskell is about *speed*—spawning threads onto multiple processors so that your program will run faster. The 'threads' are always *advisory*—if the runtime system thinks it can get the job done more quickly by sequential execution, then fine.

A Parallel Haskell program implies multiple processes running on multiple processors, under a PVM (Parallel Virtual Machine) framework. An MPI interface is under development but not fully functional, yet.

Parallel Haskell is still relatively new; it is more about "research fun" than about "speed." That will change.

Check the GPH Page (http://www.cee.hw.ac.uk/~dsg/gph/) for more information on "GPH" (Haskell98 with extensions for parallel execution), the latest version of "GUM" (the runtime system to enable parallel executions) and papers on research issues. A list of publications about GPH and about GUM is also available from Simon's Web Page.

Some details about Parallel Haskell follow. For more information about concurrent Haskell, see the module `Control.Concurrent` in the library documentation.

7.10.1. Features specific to Parallel Haskell

7.10.1.1. The `Parallel` interface (recommended)

GHC provides two functions for controlling parallel execution, through the `Parallel` interface:

```
interface Parallel where
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. At the next heap allocation, the currently executing thread will yield control to the scheduler, and the scheduler will start a new thread (until reaching the active thread limit) for each spark which has not already been evaluated to WHNF.

The expression ``(x `seq` y)`` evaluates `x` to weak head normal form and then returns `y`. The `seq` primitive can be used to force evaluation of an expression beyond WHNF, or to impose a desired execution sequence for the evaluation of an expression.

For example, consider the following parallel version of our old nemesis, `nfib`:

```
import 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.

7.10.1.2. Underlying functions and primitives

The functions `par` and `seq` are wired into GHC, and unfold into uses of the `par#` and `seq#` primitives, respectively. If you'd like to see this with your very own eyes, just run GHC with the `-ddump-simpl` option. (Anything for a good time...)

7.10.1.3. Scheduling policy for concurrent threads

Runnable threads are scheduled in round-robin fashion. Context switches are signalled by the generation of new sparks or by the expiry of a virtual timer (the timer interval is configurable with the `-C[<num>]` RTS option). However, a context switch doesn't really happen until the current heap block is full. You can't get any faster context switching than this.

When a context switch occurs, pending sparks which have not already been reduced to weak head normal form are turned into new threads. However, there is a limit to the number of active threads (runnable or blocked) which are allowed at any given time. This limit can be adjusted with the `-t<num>` RTS option (the default is 32). Once the thread limit is reached, any remaining sparks are deferred until some of the currently active threads are completed.

7.10.1.4. Scheduling policy for parallel threads

In GUM we use an unfair scheduler, which means that a thread continues to perform graph reduction until it blocks on a closure under evaluation, on a remote closure or until the thread finishes.

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://haskell.org/`. The FFI support in GHC diverges from the Addendum in the following ways:

- The routines `hs_init()`, `hs_exit()`, and `hs_set_argv()` from Chapter 6.1 of the Addendum are not supported yet.
- Syntactic forms and library functions proposed in earlier versions of the FFI are still supported for backwards compatibility.
- 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. Avoid them where possible to not compromise the portability of the resulting code.

8.1.1. Arrays

The types `ByteArray` and `MutableByteArray` may be used as basic foreign types (see FFI Addendum, Section 3.2). In C land, they map to `(char *)`.

8.1.2. 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#`.

8.2. 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()`.

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 languagem, 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 "foo_stub.h"

#include "RtsAPI.h"

extern void __stginit_Foo ( void );

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

```
  startupHaskell(argc, argv, __stginit_Foo);

  for (i = 0; i < 5; i++) {
    printf("%d\n", foo(2500));
  }

  shutdownHaskell();

  return 0;
}
```

The call to `startupHaskell()` initializes GHC's runtime system. Do NOT try to invoke any Haskell functions before calling `startupHaskell()`: strange things will undoubtedly happen.

We pass `argc` and `argv` to `startupHaskell()` so that it can separate out any arguments for the RTS (i.e. those arguments between `+RTS...-RTS`).

The third argument to `startupHaskell()` is used for initializing the Haskell modules in the program. It must be the name of the initialization function for the "top" module in the program/library - in other words, the module which directly or indirectly imports all the other Haskell modules in the program. In a standalone Haskell program this would be module `Main`, but when you are only using the Haskell code as a library it may not be. If your library doesn't have such a module, then it is straightforward to create one, purely for this initialization process. 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.

After we've finished invoking our Haskell functions, we can call `shutdownHaskell()`, which terminates the RTS. It runs any outstanding finalizers and generates any profiling or stats output that might have been requested.

The functions `startupHaskell()` and `shutdownHaskell()` may be called only once each, and only in that order.

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. Using `foreign import ccall "wrapper"` with GHC

When `foreign import ccall "wrapper"` is used in a Haskell module, The C stub file `M_stub.c` generated by GHC contains small helper functions used by the code generated for the imported wrapper, so it must be linked in to the final program. 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).

8.2.2. Using function headers

When generating C (using the `-fvia-C` directive), one can assist the C compiler in detecting type errors by using the `-#include` directive (Section 4.12.5) to provide `.h` files containing function headers.

For example,

```
#include "HsFFI.h"

void         initialiseEFS (HsInt size);
HsInt        terminateEFS (void);
HsForeignObj emptyEFS(void);
HsForeignObj updateEFS (HsForeignObj a, HsInt i, HsInt x);
HsInt        lookupEFS (HsForeignObj a, HsInt i);
```

The types `HsInt`, `HsForeignObj` etc. are described in the H98 FFI Addendum.

Note that this approach is only *essential* for returning `floats` (or if `sizeof(int) != sizeof(int *)` on your architecture) but is a Good Thing for anyone who cares about writing solid code. You're crazy not to do it.

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 1.2 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 `-fno-implicit-prelude`, 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 4.9.6.1.

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 1.2).

“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 guranteed 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 (http://www.haskell.org/happy/).

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:

`-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.

`-o FILE` or `--output=FILE`

Name of the Haskell file.

`--help`

Display a summary of the available flags.

`--version`

Output version information.

`--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`.

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.

#if condition
#ifdef name
#ifndef name
#elif condition
#else
#endif
#error message
#warning message

Conditional compilation directives are passed unmodified to the C program, C file, and C header. Putting them in the C program means that appropriate parts of the Haskell file will be skipped.

`#const C_expression`

The expression must be convertible to `long` or `unsigned long`. Its value (literal or negated literal) will be output.

`#const_str C_expression`

The expression must be convertible to const char pointer. Its value (string literal) will be output.

`#type C_type`

A Haskell equivalent of the C numeric type will be output. It will be one of `{Int,Word}{8,16,32,64}`, `Float`, `Double`, `LDouble`.

`#peek struct_type, field`

A function that peeks a field of a C struct will be output. It will have the type `Storable b => Ptr a -> IO b`. The intention is that `#peek` and `#poke` can be used for implementing the operations of class `Storable` for a given C struct (see the `Foreign.Storable` module in the library documentation).

`#poke struct_type, field`

Similarly for poke. It will have the type `Storable b => Ptr a -> b -> IO ()`.

`#ptr struct_type, field`

Makes a pointer to a field struct. It will have the type `Ptr a -> Ptr b`.

`#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 defininition in a `printf` call.

Chapter 11. Building and using Win32 DLLs

On Win32 platforms, the compiler is capable of both producing and using dynamic link libraries (DLLs) containing ghc-compiled code. This section shows you how to make use of this facility.

Until recently, **strip** didn't work reliably on DLLs, so you should test your version with care, or make sure you have the latest binutils. Unfortunately, we don't know exactly which version of binutils cured the problem (it was supposedly fixed some years ago).

11.1. Linking with DLLs

The default on Win32 platforms is to link applications in such a way that the executables will use the Prelude and system libraries DLLs, rather than contain (large chunks of) them. This is transparent at the command-line, so

```
sh$ cat main.hs
module Main where
main = putStrLn "hello, world!"
sh$ ghc -o main main.hs
ghc: module version changed to 1; reason: no old .hi file
sh$ strip main.exe
sh$ ls -l main.exe
-rwxr-xr-x   1 544      everyone     4608 May  3 17:11 main.exe*
sh$ ./main
hello, world!
sh$
```

will give you a binary as before, but the `main.exe` generated will use the Prelude and RTS DLLs instead of linking them in statically.

4K for a `"hello, world"` application—not bad, huh? :-)

11.2. Not linking with DLLs

If you want to build an executable that doesn't depend on any ghc-compiled DLLs, use the `-static` option to link in the code statically.

Notice that you cannot mix code that has been compiled with `-static` and not, so you have to use the `-static` option on all the Haskell modules that make up your application.

11.3. Creating a DLL

Making libraries into DLLs doesn't work on Windows at the moment (and is no longer supported); however, all the machinery is still there. If you're interested, contact the GHC team. Note that building an entire Haskell application as a DLL is still supported (it's just inter-DLL Haskell calls that don't work). 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 --mk-dll -o foo.dll bar.o baz.o wibble.a -lfooble
```

By feeding the ghc compiler driver the option `--mk-dll`, 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.

To create a 'static' DLL, i.e. one that does not depend on the GHC DLLs, use the `-static` when compiling up your Haskell code and building the DLL.

A couple of things to notice:

- Since DLLs correspond to packages (see Section 4.10) you need to use `-package-name dll-name` when compiling modules that belong to a DLL if you're going to call them from Haskell. Otherwise, Haskell code that calls entry points in that DLL will do so incorrectly, and crash. For similar reasons, you can only compile a single module tree into a DLL, as `startupHaskell` needs to be able to call its initialisation function, and only takes one such argument (see Section 11.4). Hence the modules you compile into a DLL must have a common root.
- By default, the entry points of all the object files will be exported from the DLL when using `--mk-dll`. Should you want to constrain this, you can specify the *module definition file* to use on the command line as follows:

  ```
  ghc --mk-dll -o .... -optdll--def -optdllMyDef.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 `--mk-dll` 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_imp.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_imp.a`.

Additionally, when the compiler driver is linking in non-static mode, it will rewrite occurrence of `-lHSfoo` on the command line to `-lHSfoo_imp`. By doing this for you, switching from non-static to static linking is simply a question of adding `-static` to your command line.

11.4. 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. The dirty details are in the *Foreign Function Interface* definition, but it can be tricky to work out how to combine this with DLL building, so 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>

  EXTFUN(__stginit_Adder);

  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 --mk-dll -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.

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

- The Haskell report specifies that programs may be written using Unicode. GHC only accepts the ISO-8859-1 character set at the moment.
- 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 `.\`.
- When `-fglasgow-exts` is on, GHC reserves several keywords beginning with two underscores. This is due to the fact that GHC uses the same lexical analyser for interface file parsing as it does for source file parsing, and these keywords are used in interface files. Do not use any identifiers beginning with a double underscore in `-fglasgow-exts` mode.

12.1.1.2. Context-free syntax

- 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

Very long `String` constants:

May not go through. If you add a "string gap" every few thousand characters, then the strings can be as long as you like.

Bear in mind that string gaps and the `-cpp` option don't mix very well (see Section 4.12.3).

12.1.1.4. Declarations and bindings

None known.

12.1.1.5. Module system and interface files

Namespace pollution

Several modules internal to GHC are visible in the standard namespace. All of these modules begin with `Prel`, so the rule is: don't use any modules beginning with `Prel` in your program, or you may be comprehensively screwed.

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)])
```

12.1.1.7. In Prelude support

The `Char` type

The Haskell report says that the `Char` type holds 16 bits. GHC follows the ISO-10646 standard a little more closely: `maxBound :: Char` in GHC is `0x10FFFF`.

Arbitrary-sized tuples:

Tuples are currently limited to size 61. HOWEVER: standard instances for tuples (`Eq`, `Ord`, `Bounded`, `Ix Read`, and `Show`) are available *only* up to 5-tuples.

This limitation is easily subvertible, so please ask if you get stuck on it.

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.

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).

12.2. Known bugs or infelicities

GHC has the following known bugs or infelicities:

- GHC only provides tuples up to size 62, and derived tuple instances (for Eq, Ord, etc) up to size 15.
- GHC can warn about non-exhaustive or overlapping patterns, 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.
- Dangers with multiple Main modules.

  GHC does not insist that module `Main` lives in a file called `Main.hs`. This is useful if you want multiple versions of `Main`. But there's a danger: when compiling module `Main` (regardless of what file it comes from), GHC looks for the interface `Main.hi`; it uses this to get version information from the last time it recompiled `Main`. The trouble is that this `Main.hi` may not correspond to the source file being compiled.

  Solution: remove `Main.hi` first. A better solution would be for GHC to record the source-file filename in the interface file, or even an MD5 checksum.

- 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).

- GHCi 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.
- 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 program, 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 (http://research.microsoft.com/~simonpj/Papers/inlining).

Chapter 13. GHC FAQ

This section has the answers to questions that get asked regularly on the GHC mailing lists, in no particular order. Please let us know if you think there's a question/answer that should be added here.

How do I port GHC to platform X?

There are two distinct possibilities: either

- The hardware architecture for your system is already supported by GHC, but you're running an OS that isn't supported (or perhaps has been supported in the past, but currently isn't). This is the easiest type of porting job, but it still requires some careful bootstrapping.
- Your system's hardware architecture isn't supported by GHC. This will be a more difficult port (though by comparison perhaps not as difficult as porting gcc).

Both ways require you to bootrap from intermediate `HC` files: these are the stylised C files generated by GHC when it compiles Haskell source. Basically the idea is to take the HC files for GHC itself to the target machine and compile them with `gcc` to get a working GHC, and go from there.

The Building Guide (http://www.haskell.org/ghc/latest/building/building-guide.html) has all the details on how to bootstrap GHC on a new platform.

Do I have to recompile all my code if I upgrade GHC?

Yes. There are two reasons for this:

- GHC does a lot of cross-module optimisation, so compiled code will include parts of the libraries it was compiled against (including the Prelude), so will be deeply tied to the actual version of those libraries it was compiled against. When you upgrade GHC, the libraries may change; even if the external interface of the libraries doesn't change, sometimes internal details may change because GHC optimised the code in the library differently.
- We sometimes change the ABI (application binary interface) between versions of GHC. Code compiled with one version of GHC is not necessarily compatible with code compiled by a different version, even if you arrange to keep the same libraries.

Why doesn't GHC use shared libraries?

The subject of shared libraries has come up several times in the past — take a look through the mailing-list archives for some of the previous discussions. The upshot is that shared libraries wouldn't really buy much unless you really need to save the disk space: in all other considerations, static linking comes out better.

Unfortunately GHC-compiled libraries are very tightly coupled, which means it's unlikely you'd be able to swap out a shared library for a newer version unless it was compiled with *exactly* the same compiler and set of libraries as the old version.

I can't get string gaps to work

If you're also using CPP, beware of the known pitfall with string gaps mentioned in Section 4.12.3.1.

GHCi complains about missing symbols like `CC_LIST` when loading a previously compiled .o file.

This probably means the .o files in question were compiled for profiling (with `-prof`). Workaround: recompile them without profiling. We really ought to detect this situation and give a proper error message.

Linking a program causes the following error on Linux: `/usr/bin/ld: cannot open -lgmp: No such file or directory`

The problem is that your system doesn't have the GMP library installed. If this is a RedHat distribution, install the RedHat-supplied `gmp-devel` package, and the `gmp` package if you don't already have it. There have been reports that installing the RedHat packages also works for SuSE (SuSE don't supply a shared gmp library).

I Can't run GHCi on Linux, because it complains about a missing `libreadline.so.3`.

The "correct" fix for this problem is to install the correct RPM for the particular flavour of Linux on your machine. If this isn't an option, however, there is a hack that might work: make a symbolic link from `libreadline.so.4` to `libreadline.so.3` in `/usr/lib`. We tried this on a SuSE 7.1 box and it seemed to work, but YMMV.

Solaris users may sometimes get link errors due to libraries needed by GNU Readline.

We suggest you try linking in some combination of the `termcap`, `curses` and `ncurses` libraries, by giving `-ltermcap`, `-lcurses` and `-lncurses` respectively. If you encounter this problem, we would appreciate feedback on it, since we don't fully understand what's going on here.

When I try to start ghci (probably one I compiled myself) it says `ghc-5.02: not built for interactive use`

To build a working ghci, you need to build GHC 5.02 with itself; the above message appears if you build it with 4.08.X, for example. It'll still work fine for batch-mode compilation, though. Note that you really must build with exactly the same version of the compiler. Building 5.02 with 5.00.2, for example, may or may not give a working interactive system; it probably won't, and certainly isn't supported. Note also that you can build 5.02 with any older compiler, back to 4.08.1, if you don't want a working interactive system; that's OK, and supported.

When I use a foreign function that takes or returns a float, it gives the wrong answer, or crashes.

You should use the `-#include` option to bring the correct prototype into scope (see Section 4.12.5).

My program that uses a really large heap crashes on Windows.

For utterly horrible reasons, programs that use more than 128Mb of heap won't work when compiled dynamically on Windows (they should be fine statically compiled).

GHC doesn't like filenames containing `+`.

Indeed not. You could change `+` to `p` or `plus`.

When I open a FIFO (named pipe) and try to read from it, I get EOF immediately.

This is a consequence of the fact that GHC opens the FIFO in non-blocking mode. The behaviour varies from OS to OS: on Linux and Solaris you can wait for a writer by doing an explicit `threadWaitRead` on the file descriptor (gotten from `Posix.handleToFd`) before the first read, but this doesn't work on FreeBSD (although rumour has it that recent versions of FreeBSD changed the behavour to match other OSs). A workaround for all systems is to open the FIFO for writing yourself, before (or at the same time as) opening it for reading.

When I `foreign import` a function that returns `char` or `short`, I get garbage back.

This is a known bug in GHC versions prior to 5.02.2. GHC doesn't mask out the more significant bits of the result. It doesn't manifest with gcc 2.95, but apparently shows up with g++ and gcc 3.0.

My program is failing with `head []`, or an array bounds error, or some other random error, and I have no idea how to find the bug. Can you help?

Compile your program with `-prof -auto-all` (make sure you have the profiling libraries installed), and run it with `+RTS -xc -RTS` to get a "stack trace" at the point at which the exception was raised. See Section 4.16.3 for more details.

How do I increase the heap size permanently for a given binary?

See Section 4.16.4.

I'm trying to compile my program for parallel execution with the `-parallel`, and GHC complains with an error like "failed to load interface file for Prelude".

GHC doesn't ship with support for parallel execution, that support is provided separately by the GPH (http://www.macs.hw.ac.uk/~dsg/gph/) project.

When is it safe to use `unsafePerformIO`?

We'll give two answers to this question, each of which may be helpful. These criteria are not rigorous in any real sense (you'd need a formal semantics for Haskell in order to give a proper answer to this question), but should give you a feel for the kind of things you can and cannot do with `unsafePerformIO`.

- It is safe to implement a function or API using `unsafePerformIO` if you could imagine also implementing the same function or API in Haskell without using `unsafePerformIO` (forget about efficiency, just consider the semantics).
- In pure Haskell, the value of a function depends only on the values of its arguments (and free variables, if it has any). If you can implement the function using `unsafePerformIO` and still retain this invariant, then you're probably using `unsafePerformIO` in a safe way. Note that you need only consider the *observable* values of the arguments and result.

For more information, see this thread (http://www.haskell.org/pipermail/glasgow-haskell-users/2002-July/003681.html).

Why does linking take so long?

Linking a small program should take no more than a few seconds. Larger programs can take longer, but even linking GHC itself only takes 3-4 seconds on our development machines.

Long link times have been attributed to using Sun's linker on Solaris, as compared to GNU **ld** which appears to be much faster. So if you're on a Sun box, try switching to GNU **ld**. This article (http://www.haskell.org/pipermail/glasgow-haskell-users/2002-November/004477.html) from the mailing list has more information.