The Glasgow Haskell Compiler User's Guide, Version 4.08

The GHC Team

The Glasgow Haskell Compiler User's Guide, Version 4.08

by The GHC Team

Table of Contents

List of Tables

The Glasgow Haskell Compiler License

Chapter 1. Introduction to GHC

This is a guide to using the Glasgow Haskell compilation (GHC) system. It is a batch compiler for the Haskell 98 language, with support for various Glasgow-only extensions. In this document, we assume that GHC has been installed at your site as `ghc`. A separate document, “Building and Installing the Glasgow Functional Programming Tools Suite”, describes how to install `ghc`.

Many people will use GHC very simply: compile some modules—`ghc -c -O Foo.hs Bar.hs`; and link them— `ghc -o wiggle -O Foo.o Bar.o`.

But if you need to do something more complicated, GHC can do that, too:

```
ghc -c -O -fno-foldr-build -dcore-lint -fvia-C -ddump-simpl Foo.lhs
```

Stay tuned—all will be revealed!

The rest of this section provide some tutorial information on batch-style compilation; if you’re familiar with these concepts already, then feel free to skip to the next section.

1.1. The (batch) compilation system components

The Glorious Haskell Compilation System, as with most UNIX (batch) compilation systems, has several interacting parts:

1. A *driver* `ghc`—which you usually think of as “the compiler”—is a program that merely invokes/glues-together the other pieces of the system (listed below), passing the right options to each, slurping in the right libraries, etc.
2. A *literate pre-processor* `unlit` that extracts Haskell code from a literate script; used if you believe in that sort of thing.
3. The *Haskellised C pre-processor* `hscpp`, only needed by people requiring conditional compilation, probably for large systems. The “Haskellised” part just means that `#line` directives in the output have been converted into proper Haskell `{-# LINE ... -}` pragmas. You must give an explicit `-cpp` option for the C pre-processor to be invoked.
4. The *Haskell compiler* `hsc`, which—in normal use—takes its input from the C pre-processor and produces assembly-language output (sometimes: ANSI C output).
5. The *ANSI C Haskell high-level assembler :-)* compiles `hsc`’s C output into assembly language for a particular target architecture. In fact, the only C compiler we currently support is `gcc`, because we make use of certain extensions to the C language only supported by gcc. Version 2.x is a must; we recommend version 2.7.2.1 for stability (we’ve heard both good and bad reports of later versions).
6. The *assembler*—a standard UNIX one, probably `as`.
7. The *linker*—a standard UNIX one, probably `ld`.

8. A *runtime system*, including (most notably) a storage manager; the linker links in the code for this.
9. The *Haskell standard prelude*, a large library of standard functions, is linked in as well.
10. Parts of other *installed libraries* that you have at your site may be linked in also.

1.2. What really happens when I “compile” a Haskell program?

You invoke the Glasgow Haskell compilation system through the driver program `ghc`. For example, if you had typed a literate “Hello, world!” program into `hello.lhs`, and you then invoked:

```
ghc hello.lhs
```

the following would happen:

1. The file `hello.lhs` is run through the literate-program code extractor `unlit`, feeding its output to
2. The Haskell compiler proper `hsc`, which produces input for
3. The assembler (or that ubiquitous “high-level assembler,” a C compiler), which produces an object file and passes it to
4. The linker, which links your code with the appropriate libraries (including the standard prelude), producing an executable program in the default output file named `a.out`.

You have considerable control over the compilation process. You feed command-line arguments (call them “options,” for short) to the driver, `ghc`; the “types” of the input files (as encoded in their names’ suffixes) also matter.

Here’s hoping this is enough background so that you can read the rest of this guide!

1.3. 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 two 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. Subscription can be done on-line at `http://www.haskell.org/mailman/listinfo/glasgow-haskell-users`.
>
> To communicate with your fellow users, send mail to <`glasgow-haskell-users@haskell.org`>.
>
> To contact the list administrator, send mail to <`glasgow-haskell-users-admin@haskell.org`>. An archive of the list is available at `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.
>
> Subscription can be done on-line at `http://www.haskell.org/mailman/listinfo/glasgow-haskell-bugs`.
>
> Again, you may contact the list administrator at <`glasgow-haskell-bugs-admin@haskell.org`>. And, yes, an archive of the list is available on the Web at the `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.)
>
> To subscribe: `http://www.haskell.org/mailman/listinfo/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`.

1.4. 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 may need to recompile any code that was compiled against the old libraries.

The value of `__GLASGOW_HASKELL__` (see Section 3.9.1) 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.

1.5. Release notes for version 4.08 (July 2000)

1.5.1. User-visible compiler changes

- In 4.08.2 (end of Jan 2001): Various minor bug-fixes, but nothing major.
- New profiling subsystem, based on cost-centre stacks. See Chapter 4.
- The x86 native code generator has been reworked considerably, and now works reliably. Using the NCG rather than compiling via C reduces compilation times by roughly a half while having minimal effect on the run-time of the compiled program (about 2-4% slower, worse for floating-point intensive programs).

  The NCG is used by default for non-optimising compiles. You can use it with `-O` by adding the `-fasm-x86` flag to GHC's command line, *after* `-O`.
- Implicit parameters. This Haskell extension gives a statically-typed version of dynamic scoping that avoids the worst problems of dynamic scoping in lisp. See the POPL paper (http://www.cse.ogi.ed/~jlewis/implicit.ps.gz) for more details. It is enabled by `-fglasgow-exts`.

- New `DEPRECATED` pragma for marking outdated interfaces as deprecated.
- New flag: `-ddump-minimal-imports`, which dumps a file `M.imports` that contains the (allegedly) minimal bunch of imports needed by the current module.
- New "package" system for libraries. See Section 3.7.4.1 for the details.
- The long-standing bug that caused some programs which used `trace` to exit with a deadlock error has been fixed.
- Trying to put into a full `MVar` will now raise a `PutFullMVar` exception.
- If a thread is about to be garbage collected, because it is waiting on an `MVar` that no other thread has access to, then it will now be sent the `BlockedOnDeadMVar` exception.
- A thread that is found to be blocked against itself (i.e. is black holed) is now sent a `NonTermination` exception.
- Operations which may *block*, such as `takeMVar`, `raiseInThread`, and several I/O operations, may now receive asynchronous exceptions even in the scope of a `blockAsyncExceptions`. These are called *interruptible* operations. See Section 4.8.7.2 in *Haskell Libraries* for more details.
- Result type signatures now work.
- A truckload of bugfixes.

1.5.2. User-visible library changes

- The FFI has been revised and expanded; see Section 4.9 in *Haskell Libraries*, Section 4.5 in *Haskell Libraries*, and Section 4.6 in *Haskell Libraries* .
- HaXml, a library for parsing and generating XML, has been added to the `text` package (Section 8.1 in *Haskell Libraries*).
- The `QuickCheck` library for performing functional testing has been added to the `util` package (Section 9.3 in *Haskell Libraries*).
- Two new experimental interfaces to arrays: `IArray` for immutable arrays (Section 4.12 in *Haskell Libraries*), and `MArray` for mutable arrays (Section 4.16 in *Haskell Libraries*). Comments on these interfaces are welcome; eventually we'd like them to replace `ByteArray`, `MutableArray`, `IOArray`, and `STArray`.
- New function: `tryTakeMVar`.
- `hPutBuf`, `hPutBufBA`, `hGetBuf`, and `hGetBufBA`, have been renamed to `hPutBufFull`, `hPutBufBAFull`, `hGetBufFull`, and `hGetBufBAFull`. Functions with the old names still exist, but have slightly different semantics. See Section 4.14.5 in *Haskell Libraries* for more details.

1.5.3. Internal changes

- `Con` is gone; the `CoreExpr` type is simpler.
- `NoRepLits` have gone.
- Better usage info in interface files, which means less recompilation.
- CCall primop is tidied up.
- Constant folding is now done by Rules.
- Lots of hackery in the simplifier.
- Improvements in CPR and strictness analysis.

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 two parts: installing on Unix-a-likes, and installing on Windows.

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 Linux/x86

> RPM source & binary packages for RedHat 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, `fptools`, 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 follows.

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 (GHC) to run on Windows platforms can be a bit of a trying experience. It should be much easier now than in the past, since all the software required to build and use GHC is included in the InstallShield.

An installation of GHC requires about 70M of disk space (which can be reduced by choosing a "compact" installation). To run GHC comfortably, your machine should have at least 64M of memory.

2.2.1. Installing GHC

Download the latest GHC distribution (ghc-4.08 InstallShield installer, 22M) from haskell.org (http://www.haskell.org/ghc/download.html) It is packaged up using an installer that should be familiar-looking to Windows users. The 4.08 InstallShield package comes with some Cygwin binaries, which unfortunately won't work with another Cygwin installation on the same machine, unless it uses exactly the same version of the Cygwin DLL (1.3.1 in this InstallShield).

Note: The Cygwin support for long file names containing spaces is not 100%, so make sure that you install ghc in a directory that has no embedded spaces (i.e., resist the temptation to put it in `/Program Files/`!)

When the installer has completed, make sure you add the location of the ghc `bin/` directory to your path (e.g. `/ghc/ghc-4.08/bin`). You need to do this in order to bring the various GHC binaries into scope. Also, if the directory `C:/TEMP` doesn't already exist, you should create it.

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$
```

OK, assuming that worked, you're all set. Go forth and write useful Haskell programs :-) If not, consult the installation FAQ (Section 2.2.2); if that still doesn't help then please report the problems you're experiencing (see Chapter 8).

Further information on using GHC under Windows can be found in Sigbjørn Finne's pages (http://www.dcs.gla.ac.uk/~sof/ghc-win32.html). Note: ignore the installation instructions, which are rather out of date; the *Miscellaneous* section at the bottom of the page is of most interest, covering topics beyond the scope of this manual.

2.2.2. 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 **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.

3. I get errors when trying to build GHC 4.08 with GHC 4.05.

This seems to work better if you don’t use `-O` in `GhcHcOpts`. It’s a bug in 4.05, unfortunately. Anyway, better to install 4.08 binaries and use those.

2.3. Building the documentation

We use the DocBook DTD, which is widely used. Most shrink-wrapped distributions seem to be broken in one way or another; thanks to heroic efforts by Sven Panne and Manuel Chakravarty, we now support most of them, plus properly installed versions.

Instructions on installing and configuring the DocBook tools follow.

2.3.1. Installing the DocBook tools from RPMs

If you’re using a system that can handle RedHat RPM packages, you can probably use the Cygnus DocBook tools (http://sourceware.cygnus.com/docbook-tools/), which is the most shrink-wrapped SGML suite that we could find. You need all the RPMs except for psgml (i.e. `docbook`, `jade`, `jadetex`, `sgmlcommon` and `stylesheets`). Note that most of these RPMs are architecture neutral, so are likely to be found in a `noarch` directory. The SuSE RPMs also work; the RedHat ones *don’t* in RedHat 6.2 (7.0 and later should be OK), but they are easy to fix: just make a symlink from `/usr/lib/sgml/stylesheets/nwalsh-modular/lib/dblib.dsl` to `/usr/lib/sgml/lib/dblib.dsl`.

2.3.2. Installing DocBook on FreeBSD

On FreeBSD systems, the easiest way to get DocBook up and running is to install it from the ports tree or a pre-compiled package (packages are available from your local FreeBSD mirror site).

To use the ports tree, do this:

```
$ cd /usr/ports/textproc/docproj
$ make install
```

This installs the FreeBSD documentation project tools, which includes everything needed to format the GHC documentation.

2.3.3. Installing from binaries on Windows

It's a good idea to use Norman Walsh's installation notes (http://nwalsh.com/docbook/dsssl/doc/install.html) as a guide. You should get version 3.1 of DocBook, and note that his file `test.sgm` won't work, as it needs version 3.0. You should unpack Jade into `\Jade`, along with the entities, DocBook into `\docbook`, and the DocBook stylesheets into `\docbook\stylesheets` (so they actually end up in `\docbook\stylesheets\docbook`).

2.3.4. Installing the DocBook tools from source

2.3.4.1. Jade

Install OpenJade (http://openjade.sourceforge.net/) (Windows binaries are available as well as sources). If you want DVI, PS, or PDF then install JadeTeX from the `dsssl` subdirectory. (If you get the error:

```
! LaTeX Error: Unknown option implicit=false' for package hyperref'.
```

your version of **hyperref** is out of date; download it from CTAN (`macros/latex/contrib/supported/hyperref`), and make it, ensuring that you have first removed or renamed your old copy. If you start getting file not found errors when making the test for **hyperref**, you can abort at that point and proceed straight to **make install**, or enter them as `../`*filename*.)

Make links from `virtex` to `jadetex` and `pdfvirtex` to `pdfjadetex` (otherwise DVI, PostScript and PDF output will not work). Copy `dsssl/*.{dtd,dsl}` and `catalog` to `/usr/[local/]lib/sgml`.

2.3.4.2. DocBook and the DocBook stylesheets

Get a Zip of DocBook (http://www.oasis-open.org/docbook/sgml/3.1/index.html) and install the contents in `/usr/[local/]/lib/sgml`.

Get the DocBook stylesheets (http://nwalsh.com/docbook/dsssl/) and install in `/usr/[local/]lib/sgml/stylesheets` (thereby creating a subdirectory docbook). For indexing, copy or link `collateindex.pl` from the DocBook stylesheets archive in `bin` into a directory on your `PATH`.

Download the ISO entities (http://www.oasis-open.org/cover/ISOEnts.zip) into `/usr/[local/]lib/sgml`.

2.3.5. Configuring the DocBook tools

Once the DocBook tools are installed, the configure script will detect them and set up the build system accordingly. If you have a system that isn't supported, let us know, and we'll try to help.

2.3.6. Remaining problems

If you install from source, you'll get a pile of warnings of the form

```
DTDDECL catalog entries are not supported
```

every time you build anything. These can safely be ignored, but if you find them tedious you can get rid of them by removing all the `DTDDECL` entries from `docbook.cat`.

Chapter 3. Using GHC

GHC is a command-line compiler: in order to compile a Haskell program, GHC must be invoked on the source file(s) by typing a command to the shell. The steps involved in compiling a program can be automated using the **make** tool (this is especially useful if the program consists of multiple source files which depend on each other). This section describes how to use GHC from the command-line.

3.1. Overall command-line structure

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`. For conflicting options, e.g., `-c -S`, we reserve the right to do anything we want. (Usually, the last one applies.)

3.2. 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.

3.3. Help and verbosity options

A good option to start with is the `-help` (or `-?`) option. GHC spews a long message to standard output and then exits.

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.

If you're just interested in the compiler version number, the `-version` option prints out a one-line string containing the requested info.

3.4. Running the right phases in the right order

The basic task of the **ghc** driver is to run each input file through the right phases (compiling, linking, etc.).

The first phase to run is determined by the 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	-	-
C pre-processor (opt.)	-	-	-
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).

The option `-cpp` must be given for the C pre-processor phase to be run, that is, the pre-processor will be run over your Haskell source file before continuing.

The option `-E` runs just the pre-processing passes of the compiler, outputting the result on stdout before stopping. If used in conjunction with -cpp, the output is the code blocks of the original (literal) source after having put it through the grinder that is the C pre-processor. Sans `-cpp`, the output is the de-litted version of the original source.

The option `-optcpp-E` runs just the pre-processing stage of the C-compiling phase, sending the result to stdout. (For debugging or obfuscation contests, usually.)

3.5. Re-directing the compilation output(s)

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!

EXOTICA: But 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`.

MORE EXOTICA: The `-osuf <suffix>` will change the `.o` file suffix for object files to whatever you specify. (We use this in compiling the prelude.). Similarly, the `-hisuf <suffix>` will change the `.hi` file suffix for non-system interface files (see Section 3.7.3).

The `-hisuf`/`-osuf` game is useful if you want to compile a program with both GHC and HBC (say) in the same directory. Let HBC use the standard `.hi`/`.o` suffixes; add `-hisuf g_hi -osuf g_o` to your **make** rule for GHC compiling...

FURTHER EXOTICA: If you are doing a normal `.hs`-to-`.o` compilation but would like to hang onto the intermediate `.hc` C file, just throw in a `-keep-hc-file-too` option. If you would like to look at the assembler output, toss in a `-keep-s-file-too`, too.

3.5.1. Saving GHC's standard error output

Sometimes, you may cause GHC to be rather chatty on standard error; with `-v`, for example. You can instruct GHC to *append* this output to a particular log file with a `-odump <blah>` option.

3.5.2. Redirecting temporary files

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 `TMPDIR` variable when building GHC, and never worry about `TMPDIR` again. (see the build documentation).

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

`-Wnot`:

Turns off all warnings, including the standard ones.

`-w`:

Synonym for `-Wnot`.

`-W`:

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

`-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-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 *not* allow 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-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-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-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-unused-imports`:

Report any objects that are explicitly imported but never used.

`-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-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.

`-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-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-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.

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.)

3.7. Separate compilation

This section describes how GHC supports separate compilation.

3.7.1. Interface files

When GHC compiles a source file `F` which contains a module `A`, say, it generates an object `F.o`, *and* a companion *interface file* `A.hi`. The interface file is not intended for human consumption, as you'll see if you take a look at one. It's merely there to help the compiler compile other modules in the same program.

NOTE: Having the name of the interface file follow the module name and not the file name, means that working with tools such as **make** become harder. **make** implicitly assumes that any output files produced by processing a translation unit will have file names that can be derived from the file name of the translation unit. For instance, pattern rules becomes unusable. For this reason, we recommend you stick to using the same file name as the module name.

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`.

Furthermore, when compiling module `C` which imports `B`, GHC may decide that it needs to know something about `A`—for example, `B` might export a function that involves a type defined in `A`. In this case, GHC will go and read **A.hi** even though `C` does not explicitly import `A` at all.

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. 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.

3.7.2. 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 prepends a colon-separated list of `dirs` to the "import directories" list. See also Section 3.7.4 for the significance of using relative and absolute pathnames in the `-i` list.

`-i`

resets the "import directories" list back to nothing.

`-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. You are unlikely to get very far without a Prelude, but, hey, it's a free country.

`-package <lib>`

If you are using a system-supplied non-Prelude library (e.g., the POSIX library), just use a `-package posix` option (for example). The right interface files should then be available. The accompanying HsLibs document lists the libraries available by this mechanism.

`-I<dir>`

Once a Haskell module has been compiled to C (`.hc` file), you may wish to specify where GHC tells the C compiler to look for `.h` files. (Or, if you are using the `-cpp` option, where it tells the C pre-processor to look. . .) For this purpose, use a `-I` option in the usual C-ish way.

3.7.3. Other options related to interface files

The interface output may be directed to another file `bar2/Wurble.iface` with the option `-ohi bar2/Wurble.iface` (not recommended).

To avoid generating an interface file at all, use a `-nohi` option.

The compiler does not overwrite an existing `.hi` interface file if the new one is byte-for-byte the same as the old one; this is friendly to **make**. When an interface does change, it is often enlightening to be informed. The `-hi-diffs` option will make GHC run **diff** on the old and new `.hi` files. You can also record the difference in the interface file itself, the `-keep-hi-diffs` option takes care of that.

The `.hi` files from GHC contain "usage" information which changes often and uninterestingly. If you really want to see these changes reported, you need to use the `-hi-diffs-with-usages` option.

Interface files are normally jammed full of compiler-produced *pragmas*, which record arities, strictness info, etc. If you think these pragmas are messing you up (or you are doing some kind of weird experiment), you can tell GHC to ignore them with the `-fignore-interface-pragmas` option.

When compiling without optimisations on, the compiler is extra-careful about not slurping in data constructors and instance declarations that it will not need. If you believe it is getting it wrong and not importing stuff which you think it should, this optimisation can be turned off with `-fno-prune-tydecls` and `-fno-prune-instdecls`.

See also Section 3.9.3, which describes how the linker finds standard Haskell libraries.

3.7.4. The recompilation checker

`-recomp`

(On by default) Turn on recompilation checking. This will stop compilation early, leaving an existing `.o` file in place, if it can be determined that the module does not need to be recompiled.

`-no-recomp`

Turn off recompilation checking.

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. In our earlier example, module `C` does not import module `A` directly, yet 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 `-recomp` is given on the command line, 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.

3.7.4.1. Packages

To simplify organisation and compilation, GHC keeps libraries in *packages*. Packages are also compiled into single libraries on Unix, and DLLs on Windows. The term "package" can be used pretty much synonymously with "library", except that an application also forms a package, the Main package.

- A package is a group of modules. It 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. `std`)
- Each package is built into a single library (Unix; e.g. `libHSfoo.a`), or a single DLL (Windows; e.g. `HSfoo.dll`)
- The `-package-name foo` flag tells GHC that the module being compiled is destined for package `foo`. If this is omitted, the default package, `Main`, is assumed.
- The `-package foo` flag tells GHC to make available modules from package `foo`. It replaces `-syslib foo`, which is now deprecated.
- 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.

All of this tidies up the Prelude enormously. The Prelude and Standard Libraries are built into a single package called `std`. (This is a change; the library is now called `libHSstd.a` instead of `libHS.a`).

It is worth noting that on Windows, because each package is built as a DLL, and 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. This is not normally the case on most Unices.

3.7.5. 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 $@
        $(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.

3.7.6. 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 3.7.1 for details of 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.

Internally, GHC uses a script to generate the dependencies, called **mkdependHS**. This script has some options of its own, which you might find useful. Options can be passed directly to **mkdependHS** with GHC's `-optdep` option. For example, to generate the dependencies into a file called `.depend` instead of `Makefile`:

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

The full list of options accepted by **mkdependHS** is:

`-w`

Turn off warnings about interface file shadowing.

`-f blah`

Use `blah` as the makefile, rather than `makefile` or `Makefile`. If `blah` 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.

`-xdirs`

same as `-exclude-directory`.

`-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 prelude libraries as unstable, i.e., generate dependencies on the prelude modules used (including `Prelude`). This option is normally only used by the various system libraries. If a `-package` option is used, dependencies will also be generated on the library's interfaces.

3.7.7. 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:

```
__interface A 1 404 where
__export A TA{MkTA} ;
1 newtype TA = MkTA PrelBase.Int ;
```

The syntax is essentially the same as a normal `.hi` file (unfortunately), but you can usually tailor an existing `.hi` file to make a `.hi-boot` file.

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`.

The number "1" after "__interface A" gives the version number of module A; it is incremented whenever anything in A's interface file changes. The "404" is the version number of the interface file *syntax*; we change it when we change the syntax of interface files so that you get a better error message when you try to read an old-format file with a new-format compiler.

The number "1" at the beginning of a declaration is the *version number* of that declaration: for the purposes of `.hi-boot` files these can all be set to 1. All names must be fully qualified with the *original* module that an object comes from: for example, the reference to `Int` in the interface for `A` comes from `PrelBase`, which is a module internal to GHC's prelude. It's a pain, but that's the way it is.

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:

```
__interface A 1 404 where
__export A TA;
1 data TA
```

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

Note: This is all a temporary solution, a version of the compiler that handles mutually recursive modules properly without the manual construction of interface files, is (allegedly) in the works.

3.8. 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.

3.8.1. `-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 3.7.4).

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**

`-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`.

`-O2-for-C`:

Says to run GCC with `-O2`, which may be worth a few percent in execution speed. Don't forget `-fvia-C`, lest you use the native-code generator and bypass GCC altogether!

`-Onot`:

This option will make GHC "forget" any `-O`ish options it has seen so far. Sometimes useful; for example: **make all EXTRA_HC_OPTS=-Onot**.

`-Ofile <file>`:

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`!

At Glasgow, 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 -O2-for-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. Alternatively, just look at the `HsC_minus<blah>` lists in the GHC driver script.

3.8.2. `-f*`: platform-independent flags

Flags can be turned *off* individually. (NB: I hope you have a good reason for doing this. . .) To turn off the `-ffoo` flag, just use the `-fno-foo` flag. So, for example, you can say `-O2 -fno-strictness`, which will then drop out any running of the strictness analyser.

The options you are most likely to want to turn off are:

- `-fno-strictness` (strictness analyser, because it is sometimes slow),
- `-fno-specialise` (automatic specialisation of overloaded functions, because it can make your code bigger) (US spelling also accepted), and
- `-fno-cpr-analyse` switches off the CPR (constructed product result) analyser.

Should you wish to turn individual flags *on*, you are advised to use the `-Ofile` option, described above. Because the order in which optimisation passes are run is sometimes crucial, it's quite hard to do with command-line options.

Here are some "dangerous" optimisations you *might* want to try:

`-fvia-C`:

Compile via C, and don't use the native-code generator. (There are many cases when GHC does this on its own.) You might pick up a little bit of speed by compiling via C (e.g. for floating-point intensive code on Intel). If you use `_casm_`s (which are utterly deprecated), you probably *have* to use `-fvia-C`.

The lower-case incantation, `-fvia-c`, is synonymous.

Compiling via C will probably be slower (in compilation time) than using GHC's native code generator.

`-funfolding-interface-threshold<n>`:

(Default: 30) By raising or lowering this number, you can raise or lower the amount of pragmatic junk that gets spewed into interface files. (An unfolding has a "size" that reflects the cost in terms of "code bloat" of expanding that unfolding in another module. A bigger function would be assigned a bigger cost.)

`-funfolding-creation-threshold<n>`:

(Default: 30) This option is similar to `-funfolding-interface-threshold`, except that it governs unfoldings within a single module. 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`).

`-funfolding-con-discount<n>`:

(Default: 2) If the compiler decides that it can eliminate some computation by performing an unfolding, then this is a discount factor that it applies to the funciton size before deciding whether to unfold it or not.

OK, folks, these magic numbers '30', '8', and '2' are mildly arbitrary; they are of the "seem to be OK" variety. The '8' is the more critical one; it's what determines how eager GHC is about expanding unfoldings.

`-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.

`-fsemi-tagging`:

This option (which *does not work* with the native-code generator) tells the compiler to add extra code to test for already-evaluated values. You win if you have lots of such values during a run of your program, you lose otherwise. (And you pay in extra code space.)

We have not played with `-fsemi-tagging` enough to recommend it. (For all we know, it doesn't even work anymore… Sigh.)

3.8.3. `-m*`: 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.

`-mlong-calls`:

(HPPA machines) Means to pass the like-named option to GCC. Required for Very Big modules, maybe. (Probably means you're in trouble…)

`-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.

3.8.4. Code improvement by the C compiler.

The C compiler (GCC) is run with `-O` turned on. (It has to be, actually).

If you want to run GCC with `-O2`—which may be worth a few percent in execution speed—you can give a `-O2-for-C` option.

3.9. Options related to a particular phase

3.9.1. The C pre-processor

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<foo>`:

Define macro `<foo>` 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 3.13.2).

`-U<foo>`:

Undefine macro **<foo>** 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 `#defined` to *100n*. So, for version 4.00, it is 400.

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 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).

Options other than the above can be forced through to the C pre-processor with the `-opt` flags (see Section 3.13.2).

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.

3.9.2. Options affecting the C compiler (if applicable)

At the moment, quite a few common C-compiler options are passed on quietly to the C compilation of Haskell-compiler-generated C files. THIS MAY CHANGE. Meanwhile, options so sent are:

`-ansi`	do ANSI C (not K&R)
`-pedantic`	be so
`-dgcc-lint`	(hack) short for “make GCC very paranoid”

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
```

3.9.3. Linking and consistency-checking

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

`-l<FOO>`:

Link in a library named `lib<FOO>.a` which resides somewhere on the library directories path.

Because of the sad state of most UNIX linkers, the order of such options does matter. Thus: **ghc -lbar *.o** is almost certainly wrong, because it will search `libbar.a` *before* it has collected unresolved symbols from the `*.o` files. **ghc *.o -lbar** is probably better.

The linker will of course be informed about some GHC-supplied libraries automatically; these are:

-l equivalent	*description*
`-lHSrts,-lHSclib`	basic runtime libraries
`-lHS`	standard Prelude library
`-lHS_cbits`	C support code for standard Prelude library
`-lgmp`	GNU multi-precision library (for Integers)

`-package <name>`:

If you are using a Haskell "system library" (e.g., the POSIX library), just use the `-package posix` option, and the correct code should be linked in.

`-L<dir>`:

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

`-static`:

Tell the linker to avoid shared libraries.

`-no-link-chk` and `-link-chk`:

By default, immediately after linking an executable, GHC verifies that the pieces that went into it were compiled with compatible flags; a "consistency check". (This is to avoid mysterious failures caused by non-meshing of incompatibly-compiled programs; e.g., if one `.o` file was compiled for a parallel machine and the others weren't.) You may turn off this check with `-no-link-chk`. You can turn it (back) on with `-link-chk` (the default).

`-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 the ghc driver script 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.

3.10. Using Concurrent Haskell

GHC (as of version 4.00) 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 (i.e. `Concurrent` and friends), use the `-package concurrent` option.

Three RTS options are provided for modifying the behaviour of the threaded runtime system. See the descriptions of `-C[<us>]`, `-q`, and `-t<num>` in Section 3.11.4.

Concurrent Haskell is described in more detail in Chapter 2 in *Haskell Libraries*.

3.11. 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 `-N<n>`, to say how many PVM “processors” your program to run on. (For more details of all relevant RTS options, please see Section 3.11.4.)

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.

3.11.1. 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.

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**!

3.11.2. 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 `-q` 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 -N8 -q
% 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/`.

3.11.3. 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**.

3.11.4. RTS options for Concurrent/Parallel Haskell

Besides the usual runtime system (RTS) options (Section 3.12), there are a few options particularly for concurrent/parallel execution.

`-N<N>`:

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

`-C[<s>]`:

Sets the context switch interval to `<s>` seconds. A context switch will occur at the next heap block allocation after the timer expires (a heap block allocation occurs every 4k of allocation). With `-C0` or `-C`, context switches will occur as often as possible (at every heap block allocation). By default, context switches occur every 20ms 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`).

`-t<num>`:

(PARALLEL ONLY) Limit 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.)

`-d`:

(PARALLEL ONLY) Turn on debugging. It pops up one xterm (or GDB, or something. . .) per PVM processor. We use the standard **debugger** script that comes with PVM3, but we sometimes meddle with the **debugger2** script. We include ours in the GHC distribution, in `ghc/utils/pvm/`.

`-e<num>`:

(PARALLEL ONLY) Limit 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.

`-Q<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.

3.12. 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: to send RTS options to the compiler itself, you need to prefix the option with `-optCrts`, eg. to increase the maximum heap size for a compilation to 128M, you would add `-optCrts-M128m` to the command line. The compiler understands some options directly without needing `-optCrts`: these are `-H` and `-K`.

3.12.1. RTS options to control the garbage-collector

There are several options to give you precise control over garbage collection. Hopefully, you won't need any of these in normal operation, but there are several things that can be tweaked for maximum performance.

`-A<size>`:

[Default: 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).

`-F<factor>`:

[Default: 2] This option controls the amount of memory reserved for the older generations (and in the case of a two space collector the size of the allocation area) as a factor of the amount of live data. For example, if there was 2M of live data in the oldest generation when we last collected it, then by default we'll wait until it grows to 4M before collecting it again.

The default seems to work well here. If you have plenty of memory, it is usually better to use `-H<size>` than to increase `-F<factor>`.

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

`-G<generations>`:

[Default: 2] Set the number of generations used by the garbage collector. The default of 2 seems to be good, but the garbage collector can support any number of generations. Anything larger than about 4 is probably not a good idea unless your program runs for a *long* time, because the oldest generation will never 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).

`-H<size>`:

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

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

`-k<size>`:

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

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

`-K<size>`:

[Default: 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.

`-m<n>`:

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

`-M<size>`:

[Default: 64M] 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.

`-s<file>` or `-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.

3.12.2. RTS options for profiling and Concurrent/Parallel Haskell

The RTS options related to profiling are described in Section 4.5; and those for concurrent/parallel stuff, in Section 3.11.4.

3.12.3. 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. . . "

`-r<file>`:

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

"Ticky-ticky" statistics are counts of various program actions (updates, enters, etc.) The program must have been compiled using `-ticky` (a.k.a. "ticky-ticky profiling"), and, for it to be really useful, linked with suitable system libraries. Not a trivial undertaking: consult the installation guide on how to set things up for easy "ticky-ticky" profiling. For more information, see Section 4.7.

`-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.

`-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.)

3.12.4. “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 `-H8m -K1m`:

```
#include "Rts.h"
#include "RtsFlags.h"
void defaultsHook (void) {
   RTSflags.GcFlags.stksSize =  1000002 / sizeof(W_);
   RTSflags.GcFlags.heapSize =  8000002 / sizeof(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 <stdio.h>
#define W_ unsigned long int
#define I_ long int

void
ErrorHdrHook (FILE *where)
{
    fprintf(where, "\n"); /* no "Fail: " */
}

void
OutOfHeapHook (W_ request_size, W_ heap_size) /* both sizes in bytes */
{
    fprintf(stderr, "GHC's heap exhausted;\nwhile trying to
        allocate %lu bytes in a %lu-byte heap;\nuse the '-H<size>'
        option to increase the total heap size.\n",
        request_size,
        heap_size);
}

void
StackOverflowHook (I_ stack_size)    /* in bytes */
{
    fprintf(stderr, "GHC stack-space overflow: current size
        %ld bytes.\nUse the '-K<size>' option to increase it.\n",
        stack_size);
}

void
PatErrorHdrHook (FILE *where)
{
    fprintf(where, "\n*** Pattern-matching error within GHC!\n\n
        This is a compiler bug; please report it to
        glasgow-haskell-bugs@haskell.org.\n\nFail: ");
}

void
PreTraceHook (FILE *where)
{
    fprintf(where, "\n"); /* not "Trace On" */
}

void
PostTraceHook (FILE *where)
{
    fprintf(where, "\n"); /* not "Trace Off" */
}
```

3.13. Debugging the compiler

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

3.13.1. 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 driver **ghc** has wired into it. For example, you might want to try a different assembler. The `-pgm<phase-code><program-name>` option to **ghc** will cause it to use `<program-name>` for phase `<phase-code>`, where the codes to indicate the phases are:

code	*phase*
L	literate pre-processor
P	C pre-processor (if -cpp only)
C	Haskell compiler
c	C compiler
a	assembler
l	linker
dep	Makefile dependency generator

3.13.2. Forcing options to a particular phase.

The preceding sections describe driver options that are mostly applicable to one particular phase. You may also *force* a specific option `<option>` to be passed to a particular phase `<phase-code>` by feeding the driver the option `-opt<phase-code><option>`. The codes to indicate the phases are the same as in the previous section.

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).

Besides getting options to the Haskell compiler with `-optC<blah>`, you can get options through to its runtime system with `-optCrts<blah>`.

So, for example: when I want to use my normal driver but with my profiled compiler binary, I use this script:

```
#! /bin/sh
exec /local/grasp_tmp3/simonpj/ghc-BUILDS/working-alpha/ghc/driver/ghc \
     -pgmC/local/grasp_tmp3/simonpj/ghc-BUILDS/working-hsc-prof/hsc \
     -optCrts-i0.5 \
     -optCrts-PT \
     "$@"
```

3.13.3. Dumping out compiler intermediate structures

`-noC`:

Don't bother generating C output *or* an interface file. Usually used in conjunction with one or more of the `-ddump-*` options; for example: **ghc -noC -ddump-simpl Foo.hs**

`-hi`:

Do generate an interface file. This would normally be used in conjunction with `-noC`, which turns off interface generation; thus: `-noC -hi`.

`-dshow-passes`:

Prints a message to stderr as each pass starts. Gives a warm but undoubtedly misleading feeling that GHC is telling you what's happening.

`-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-usagesp`:

UsageSP inference pre-inf and output

`-ddump-cpranal`:

CPR analyser output

`-ddump-stranal`:

strictness analyser output

`-ddump-workwrap`:

worker/wrapper split output

`-ddump-occur-anal`:

‘occurrence analysis’ output

`-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-asm`:

assembly language from the native-code generator

`-dverbose-simpl` and `-dverbose-stg`:

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-{user,debug}`:

Debugging output is in one of several "styles." Take the printing of types, for example. In the "user" style, 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).

`-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-raw-asm`:

Dump out the assembly-language stuff, before the "mangler" gets it.

`-ddump-rn-trace`:

Make the renamer be *real* chatty about what it is upto.

`-dshow-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.

3.13.4. Checking for consistency

`-dcore-lint`:

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

`-dstg-lint`:

Ditto for STG level.

`-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.

3.13.5. How to read Core syntax (from some `-ddump-*` flags)

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

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

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

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

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

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

_NI_

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

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

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

-# The ‘+’ class method (Unique: r3JH) selects the addition code
-# from a ‘Num’ dictionary (now an explicit 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.)

3.13.6. 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 (Glasgow) Haskell source file uses `casm`s, the C back-end often needs to be told about which header files to include. Rather than maintaining the list of files the source depends on in a `Makefile` (using the `-#include` command-line option), it is possible to do this directly in the source file using the `OPTIONS` pragma :

```
{-# OPTIONS -#include "foo.h" #-}
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).

Chapter 4. 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 4.4
2. Run your program with one of the profiling options `-p` or `-h`. 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.

4.1. 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  (Fi-
nal)

           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         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             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 4.3) will be able to display real loops in the call-graph.

4.1.1. 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.

4.1.2. 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.

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`>.

4.2. 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, compile it as before, but this time run it with the `-h` runtime option. This generates a file `<prog>.hp` file, which you then process with **hp2ps** to produce a Postscript file `<prog>.ps`. The Postscript file can be viewed with something like **ghostview**, or printed out on a Postscript-compatible printer.

For the RTS options that control the kind of heap profile generated, see Section 4.5. Details on the usage of the **hp2ps** program are given in Section 4.6

4.3. 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.

To run **ghcprof**, you need daVinci installed, which can be obtained from *The Graph Visualisation Tool daVinci* (http://www.tzi.de/~davinci/). Install one of the binary distributions[2], 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).

4.4. Compiler options for profiling

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

Without a `-prof` option, your `_scc_`s are ignored; so you can compiled `_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.

4.5. Runtime options for profiling

It isn't enough to compile your program for profiling with `-prof`!

When you *run* your profiled program, you must tell the runtime system (RTS) what you want to profile (e.g., time and/or space), and how you wish the collected data to be reported. You also may wish to set the sampling interval used in time profiling.

Executive summary: **./a.out +RTS -pT** produces a time profile in `a.out.prof`; **./a.out +RTS -hC** produces space-profiling info which can be mangled by **hp2ps** and viewed with **ghostview** (or equivalent).

Profiling runtime flags are passed to your program between the usual `+RTS` and `-RTS` options.

`-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 4.3.

`-i<secs>`:

Set the profiling (sampling) interval to `<secs>` seconds (the default is 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.

`-h<break-down>`:

Produce a detailed *heap profile* of the heap occupied by live closures. The profile is written to the file `<program>.hp` from which a PostScript graph can be produced using **hp2ps** (see Section 4.6).

The heap space profile may be broken down by different criteria:

`-hC`:

cost centre which produced the closure (the default).

`-hM`:

cost centre module which produced the closure.

`-hD`:

closure description—a string describing the closure.

`-hY`:

closure type—a string describing the closure's type.

`-hx`:

The `-hx` option generates heap profiling information in the XML format understood by our new profiling tool (NOTE: heap profiling with the new tool is not yet working! Use **hp2ps**-style heap profiling for the time being).

4.6. 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.

4.7. 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 3.12.

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 ad-
min, 6098829 goods, 1232595 slop)
```

```
                                to-
tal words:        2     3     4     5    6+
  69647 (  1.8%) function val-
ues                 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 val-
ues                     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 applica-
tions             0.0   0.0   0.0 100.0   0.0
      2 (  0.0%) thread state ob-
jects             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
UPD_PAP_IN_NEW:     34825:       0      0      0  34825      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. daVinci is sadly not open-source :-(.

Chapter 5. Advice on: sooner, faster, smaller, stingier

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

5.1. 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 `-Rgc-stats` option, you’ll get a garbage-collector report. (Again, you can use the cheap-and-nasty `-optCrts-Sstderr` 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 `-dshow-passes` 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).

Avoid the consistency-check on linking:

Use `-no-link-chk`; saves effort. This is probably safe in a I-only-compile-things-one-way setup.

Explicit `import` declarations:

Instead of saying `import Foo`, say `import Foo (...stuff I want...)`.

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.

5.2. 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 4. 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:

> Even with `-O`, GHC tries to use a native-code generator, if available. But the native code-generator is designed to be quick, not mind-bogglingly clever. Better to let GCC have a go, as it tries much harder on register allocation, etc.
>
> So, when we want very fast code, we use: `-O -fvia-C -O2-for-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 6.11.3 and Section 6.11.4.
>
> "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 6.11.1.

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 6.1.1 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.

Section 6.5 describes the foreign calling interface.

Don't use `Floats`:

We don't provide specialisations of Prelude functions for `Float` (but we do for `Double`). If you end up executing overloaded code, you will lose on performance, perhaps badly.

`Floats` (probably 32-bits) are almost always a bad idea, anyway, unless you Really Know What You Are Doing. Use Doubles. There's rarely a speed disadvantage—modern machines will use the same floating-point unit for both. With `Doubles`, you are much less likely to hang yourself with numerical errors.

One time when `Float` might be a good idea is if you have a *lot* of them, say a giant array of `Floats`. They take up half the space in the heap compared to `Doubles`. However, this isn't true on a 64-bit machine.

Use 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 3.12.1).

5.3. 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 specialiised 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.

5.4. Stingier: 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 4) 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 6. 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!

Executive summary of our extensions:

Unboxed types and primitive operations:

> You can get right down to the raw machine types and operations; included in this are "primitive arrays" (direct access to Big Wads of Bytes). Please see Section 6.1.1 and following.

Multi-parameter type classes:

> GHC's type system supports extended type classes with multiple parameters. Please see Section 6.6.

Local universal quantification:

> GHC's type system supports explicit universal quantification in constructor fields and function arguments. This is useful for things like defining `runST` from the state-thread world. See Section 6.7.

Extistentially quantification in data types:

> Some or all of the type variables in a datatype declaration may be *existentially quantified*. More details in Section 6.8.

Scoped type variables:

> Scoped type variables enable the programmer to supply type signatures for some nested declarations, where this would not be legal in Haskell 98. Details in Section 6.10.

Pattern guards

> Instead of being a boolean expression, a guard is a list of qualifiers, exactly as in a list comprehension. See Section 6.4.

Foreign calling:

> Just what it sounds like. We provide *lots* of rope that you can dangle around your neck. Please see Chapter 7.

Pragmas

> Pragmas are special instructions to the compiler placed in the source file. The pragmas GHC supports are described in Section 6.11.

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. Details in Section 6.12.

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. See *Haskell Libraries*.

6.1. Unboxed types and primitive operations

This module defines all the types which are primitive in Glasgow Haskell, and the operations provided for them.

6.1.1. 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.

6.1.2. Unboxed Tuples

Unboxed tuples aren’t really exported by `PrelGHC`, 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.

6.1.3. Character and numeric types

There are the following obvious primitive types:

```
type Char#
type Int#
type Word#
type Addr#
type Float#
type Double#
type Int64#
type Word64#
```

If you really want to know their exact equivalents in C, see `ghc/includes/StgTypes.h` in the GHC source tree.

Literals for these types may be written as follows:

```
1#              an Int#
1.2#            a Float#
1.34##          a Double#
'a'#            a Char#; for weird characters, use e.g. '\o<octal>'#
"a"#            an Addr# (a ‘char *’); only characters '\0'..'\255' al-
lowed
```

6.1.4. Comparison operations

```
{>,>=,==,/=,<,<=}# :: Int# -> Int# -> Bool

{gt,ge,eq,ne,lt,le}Char# :: Char# -> Char# -> Bool
    - ditto for Word# and Addr#
```

6.1.5. Primitive-character operations

```
ord# :: Char# -> Int#
chr# :: Int# -> Char#
```

6.1.6. Primitive-`Int` operations

```
{+,-,*,quotInt,remInt,gcdInt}# :: Int# -> Int# -> Int#
negateInt# :: Int# -> Int#

iShiftL#, iShiftRA#, iShiftRL# :: Int# -> Int# -> Int#
        - shift left, right arithmetic, right logical

addIntC#, subIntC#, mulIntC# :: Int# -> Int# -> (# Int#, Int# #)
- add, subtract, multiply with carry
```

Note: No error/overflow checking!

6.1.7. Primitive-`Double` and `Float` operations

```
{+,-,*,/}##         :: Double# -> Double# -> Double#
{<,<=,==,/=,>=,>}## :: Double# -> Double# -> Bool
negateDouble#       :: Double# -> Double#
double2Int#         :: Double# -> Int#
int2Double#         :: Int#    -> Double#

{plus,minux,times,divide}Float# :: Float# -> Float# -> Float#
{gt,ge,eq,ne,lt,le}Float# :: Float# -> Float# -> Bool
negateFloat#        :: Float# -> Float#
float2Int#          :: Float# -> Int#
int2Float#          :: Int#   -> Float#
```

And a full complement of trigonometric functions:

```
expDouble#      :: Double# -> Double#
logDouble#      :: Double# -> Double#
sqrtDouble#     :: Double# -> Double#
sinDouble#      :: Double# -> Double#
cosDouble#      :: Double# -> Double#
tanDouble#      :: Double# -> Double#
asinDouble#     :: Double# -> Double#
acosDouble#     :: Double# -> Double#
atanDouble#     :: Double# -> Double#
sinhDouble#     :: Double# -> Double#
coshDouble#     :: Double# -> Double#
tanhDouble#     :: Double# -> Double#
powerDouble#    :: Double# -> Double# -> Double#
```

similarly for `Float#`.

There are two coercion functions for `Float#`/`Double#`:

```
float2Double#   :: Float# -> Double#
double2Float#   :: Double# -> Float#
```

The primitive version of `decodeDouble` (`encodeDouble` is implemented as an external C function):

```
decodeDouble#   :: Double# -> PrelNum.ReturnIntAndGMP
```

(And the same for `Float#`s.)

6.1.8. Operations on/for `Integers` (interface to GMP)

We implement `Integers` (arbitrary-precision integers) using the GNU multiple-precision (GMP) package (version 2.0.2).

The data type for `Integer` is either a small integer, represented by an `Int`, or a large integer represented using the pieces required by GMP's `MP_INT` in `gmp.h` (see `gmp.info` in `ghc/includes/runtime/gmp`). It comes out as:

```
data Integer = S# Int#             - small integers
             | J# Int# ByteArray#  - large integers
```

The primitive ops to support large `Integers` use the “pieces” of the representation, and are as follows:

```
negateInteger#  :: Int# -> ByteArray# -> Integer

{plus,minus,times}Integer#, gcdInteger#,
  quotInteger#, remInteger#, divExactInteger#
:: Int# -> ByteArray#
        -> Int# -> ByteArray#
        -> (# Int#, ByteArray# #)

cmpInteger#
:: Int# -> ByteArray#
        -> Int# -> ByteArray#
        -> Int# - -1 for <; 0 for ==; +1 for >

cmpIntegerInt#
:: Int# -> ByteArray#
        -> Int#
        -> Int# - -1 for <; 0 for ==; +1 for >

gcdIntegerInt# ::
:: Int# -> ByteArray#
        -> Int#
        -> Int#

divModInteger#, quotRemInteger#
        :: Int# -> ByteArray#
        -> Int# -> ByteArray#
        -> (# Int#, ByteArray#,
                  Int#, ByteArray# #)

integer2Int# :: Int# -> ByteArray# -> Int#

int2Integer#  :: Int#  -> Integer - NB: no error-checking on these two!
word2Integer# :: Word# -> Integer

addr2Integer# :: Addr# -> Integer
        - the Addr# is taken to be a ‘char *’ string
        - to be converted into an Integer.
```

6.1.9. Words and addresses

A `Word#` is used for bit-twiddling operations. It is the same size as an `Int#`, but has no sign nor any arithmetic operations.

```
type Word#      - Same size/etc as Int# but *unsigned*
type Addr#      -
A pointer from outside the "Haskell world" (from C, probably);
                - described under "arrays"
```

`Word#`s and `Addr#`s have the usual comparison operations. Other unboxed-`Word` ops (bit-twiddling and coercions):

```
{gt,ge,eq,ne,lt,le}Word# :: Word# -> Word# -> Bool

and#, or#, xor# :: Word# -> Word# -> Word#
        - standard bit ops.

quotWord#, remWord# :: Word# -> Word# -> Word#
        - word (i.e. unsigned) versions are different from int
        - versions, so we have to provide these explicitly.

not# :: Word# -> Word#

shiftL#, shiftRL# :: Word# -> Int# -> Word#
        - shift left, right logical

int2Word#       :: Int#  -> Word# - just a cast, really
```

```
word2Int#       :: Word# -> Int#
```

Unboxed-`Addr` ops (C casts, really):

```
{gt,ge,eq,ne,lt,le}Addr# :: Addr# -> Addr# -> Bool

int2Addr#       :: Int#  -> Addr#
addr2Int#       :: Addr# -> Int#
addr2Integer#   :: Addr# -> (# Int#, ByteArray# #)
```

The casts between `Int#`, `Word#` and `Addr#` correspond to null operations at the machine level, but are required to keep the Haskell type checker happy.

Operations for indexing off of C pointers (`Addr#`s) to snatch values are listed under “arrays”.

6.1.10. Arrays

The type `Array# elt` is the type of primitive, unpointed arrays of values of type `elt`.

```
type Array# elt
```

`Array#` is more primitive than a Haskell array—indeed, the Haskell `Array` interface is implemented using `Array#`—in that an `Array#` is indexed only by `Int#`s, starting at zero. It is also more primitive by virtue of being unboxed. That doesn’t mean that it isn’t a heap-allocated object—of course, it is. Rather, being unboxed means that it is represented by a pointer to the array itself, and not to a thunk which will evaluate to the array (or to bottom). The components of an `Array#` are themselves boxed.

The type `ByteArray#` is similar to `Array#`, except that it contains just a string of (non-pointer) bytes.

```
type ByteArray#
```

Arrays of these types are useful when a Haskell program wishes to construct a value to pass to a C procedure. It is also possible to use them to build (say) arrays of unboxed characters for internal use in a Haskell program. Given these uses, `ByteArray#` is deliberately a bit vague about the type of its components. Operations are provided to extract values of type `Char#`, `Int#`, `Float#`, `Double#`, and `Addr#` from arbitrary offsets within a `ByteArray#`. (For type `Foo#`, the $i$th offset gets you the $i$th `Foo#`, not the `Foo#` at byte-position $i$. Mumble.) (If you want a `Word#`, grab an `Int#`, then coerce it.)

Lastly, we have static byte-arrays, of type `Addr#` [mentioned previously]. (Remember the duality between arrays and pointers in C.) Arrays of this types are represented by a pointer to an array in the world outside Haskell, so this pointer is not followed by the garbage collector. In other respects they are just like `ByteArray#`. They are only needed in order to pass values from C to Haskell.

6.1.11. Reading and writing

Primitive arrays are linear, and indexed starting at zero.

The size and indices of a `ByteArray#`, `Addr#`, and `MutableByteArray#` are all in bytes. It's up to the program to calculate the correct byte offset from the start of the array. This allows a `ByteArray#` to contain a mixture of values of different type, which is often needed when preparing data for and unpicking results from C. (Umm…not true of indices…WDP 95/09)

Should we provide some `sizeOfDouble#` constants?

Out-of-range errors on indexing should be caught by the code which uses the primitive operation; the primitive operations themselves do *not* check for out-of-range indexes. The intention is that the primitive ops compile to one machine instruction or thereabouts.

We use the terms "reading" and "writing" to refer to accessing *mutable* arrays (see Section 6.1.14), and "indexing" to refer to reading a value from an *immutable* array.

Immutable byte arrays are straightforward to index (all indices in bytes):

```
indexCharArray#   :: ByteArray# -> Int# -> Char#
indexIntArray#    :: ByteArray# -> Int# -> Int#
indexAddrArray#   :: ByteArray# -> Int# -> Addr#
indexFloatArray#  :: ByteArray# -> Int# -> Float#
indexDoubleArray# :: ByteArray# -> Int# -> Double#

indexCharOffAddr#   :: Addr# -> Int# -> Char#
indexIntOffAddr#    :: Addr# -> Int# -> Int#
indexFloatOffAddr#  :: Addr# -> Int# -> Float#
indexDoubleOffAddr# :: Addr# -> Int# -> Double#
indexAddrOffAddr#   :: Addr# -> Int# -> Addr#
 - Get an Addr# from an Addr# offset
```

The last of these, `indexAddrOffAddr#`, extracts an `Addr#` using an offset from another `Addr#`, thereby providing the ability to follow a chain of C pointers.

Something a bit more interesting goes on when indexing arrays of boxed objects, because the result is simply the boxed object. So presumably it should be entered—we never usually return an unevaluated object! This is a pain: primitive ops aren't supposed to do complicated things like enter objects. The current solution is to return a single element unboxed tuple (see Section 6.1.2).

```
indexArray#       :: Array# elt -> Int# -> (# elt #)
```

6.1.12. The state type

The primitive type `State#` represents the state of a state transformer. It is parameterised on the desired type of state, which serves to keep states from distinct threads distinct from one another. But the *only* effect of this parameterisation is in the type system: all values of type `State#` are represented in the same way. Indeed, they are all represented by nothing at all! The code generator "knows" to generate no code, and allocate no registers etc, for primitive states.

```
type State# s
```

The type `GHC.RealWorld` is truly opaque: there are no values defined of this type, and no operations over it. It is "primitive" in that sense - but it is *not unlifted!* Its only role in life is to be the type which distinguishes the `IO` state transformer.

```
data RealWorld
```

6.1.13. State of the world

A single, primitive, value of type `State# RealWorld` is provided.

```
realWorld# :: State# RealWorld
```

(Note: in the compiler, not a `PrimOp`; just a mucho magic `Id`. Exported from `GHC`, though).

6.1.14. Mutable arrays

Corresponding to `Array#` and `ByteArray#`, we have the types of mutable versions of each. In each case, the representation is a pointer to a suitable block of (mutable) heap-allocated storage.

```
type MutableArray# s elt
type MutableByteArray# s
```

6.1.14.1. Allocation

Mutable arrays can be allocated. Only pointer-arrays are initialised; arrays of non-pointers are filled in by "user code" rather than by the array-allocation primitive. Reason: only the pointer case has to worry about GC striking with a partly-initialised array.

```
newArray#       :: Int# -> elt -> State# s -
> (# State# s, MutableArray# s elt #)

newCharArray#   :: Int# -> State# s -
> (# State# s, MutableByteArray# s elt #)
newIntArray#    :: Int# -> State# s -
> (# State# s, MutableByteArray# s elt #)
newAddrArray#   :: Int# -> State# s -
> (# State# s, MutableByteArray# s elt #)
```

```
newFloatArray#  :: Int# -> State# s -
> (# State# s, MutableByteArray# s elt #)
newDoubleArray# :: Int# -> State# s -
> (# State# s, MutableByteArray# s elt #)
```

The size of a `ByteArray#` is given in bytes.

6.1.14.2. Reading and writing

```
readArray#       :: MutableArray# s elt -> Int# -> State# s -
> (# State# s, elt #)
readCharArray#   :: MutableByteArray# s -> Int# -> State# s -
> (# State# s, Char# #)
readIntArray#    :: MutableByteArray# s -> Int# -> State# s -
> (# State# s, Int# #)
readAddrArray#   :: MutableByteArray# s -> Int# -> State# s -
> (# State# s, Addr# #)
readFloatArray#  :: MutableByteArray# s -> Int# -> State# s -
> (# State# s, Float# #)
readDoubleArray# :: MutableByteArray# s -> Int# -> State# s -
> (# State# s, Double# #)

writeArray#       :: MutableArray# s elt -> Int# -> elt     -> State# s -
> State# s
writeCharArray#   :: MutableByteArray# s -> Int# -> Char#   -> State# s -
> State# s
writeIntArray#    :: MutableByteArray# s -> Int# -> Int#    -> State# s -
> State# s
writeAddrArray#   :: MutableByteArray# s -> Int# -> Addr#   -> State# s -
> State# s
writeFloatArray#  :: MutableByteArray# s -> Int# -> Float#  -> State# s -
> State# s
writeDoubleArray# :: MutableByteArray# s -> Int# -> Double# -> State# s -
> State# s
```

6.1.14.3. Equality

One can take “equality” of mutable arrays. What is compared is the *name* or reference to the mutable array, not its contents.

```
sameMutableArray#     :: MutableArray# s elt -> MutableArray# s elt -
> Bool
```

```
sameMutableByteArray# :: MutableByteArray# s -> MutableByteArray# s -
> Bool
```

6.1.14.4. Freezing mutable arrays

Only unsafe-freeze has a primitive. (Safe freeze is done directly in Haskell by copying the array and then using `unsafeFreeze`.)

```
unsafeFreezeArray#     :: MutableArray# s elt -> State# s -
> (# State# s, Array# s elt #)
unsafeFreezeByteArray# :: MutableByteArray# s -> State# s -
> (# State# s, ByteArray# #)
```

6.1.15. Synchronizing variables (M-vars)

Synchronising variables are the primitive type used to implement Concurrent Haskell's MVars (see the Concurrent Haskell paper for the operational behaviour of these operations).

```
type MVar# s elt        - primitive

newMVar#    :: State# s -> (# State# s, MVar# s elt #)
takeMVar#   :: SynchVar# s elt -> State# s -> (# State# s, elt #)
putMVar#    :: SynchVar# s elt -> State# s -> State# s
```

6.2. Primitive state-transformer monad

This monad underlies our implementation of arrays, mutable and immutable, and our implementation of I/O, including "C calls".

The `ST` library, which provides access to the `ST` monad, is described in Section 4.21 in *Haskell Libraries*.

6.3. Primitive arrays, mutable and otherwise

GHC knows about quite a few flavours of Large Swathes of Bytes.

First, GHC distinguishes between primitive arrays of (boxed) Haskell objects (type `Array# obj`) and primitive arrays of bytes (type `ByteArray#`).

Second, it distinguishes between...

Immutable:

> Arrays that do not change (as with "standard" Haskell arrays); you can only read from them. Obviously, they do not need the care and attention of the state-transformer monad.

Mutable:

> Arrays that may be changed or "mutated." All the operations on them live within the state-transformer monad and the updates happen *in-place*.

"Static" (in C land):

> A C routine may pass an `Addr#` pointer back into Haskell land. There are then primitive operations with which you may merrily grab values over in C land, by indexing off the "static" pointer.

"Stable" pointers:

> If, for some reason, you wish to hand a Haskell pointer (i.e., *not* an unboxed value) to a C routine, you first make the pointer "stable," so that the garbage collector won't forget that it exists. That is, GHC provides a safe way to pass Haskell pointers to C.
>
> Please see Section 4.24 in *Haskell Libraries* for more details.

"Foreign objects":

> A "foreign object" is a safe way to pass an external object (a C-allocated pointer, say) to Haskell and have Haskell do the Right Thing when it no longer references the object. So, for example, C could pass a large bitmap over to Haskell and say "please free this memory when you're done with it."
>
> Please see Section 4.10 in *Haskell Libraries* for more details.

The libraries documentatation gives more details on all these "primitive array" types and the operations on them.

6.4. 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.

6.5. The foreign interface

The foreign interface consists of the following components:

- The Foreign Function Interface language specification (included in this manual, in Chapter 7).
- The `Foreign` module (see Section 4.9 in *Haskell Libraries*) collects together several interfaces which are useful in specifying foreign language interfaces, including the following:
  - The `ForeignObj` module (see Section 4.10 in *Haskell Libraries*), for managing pointers from Haskell into the outside world.
  - The `StablePtr` module (see Section 4.24 in *Haskell Libraries*), for managing pointers into Haskell from the outside world.
  - The `CTypes` module (see Section 4.5 in *Haskell Libraries*) gives Haskell equivalents for the standard C datatypes, for use in making Haskell bindings to existing C libraries.
  - The `CTypesISO` module (see Section 4.6 in *Haskell Libraries*) gives Haskell equivalents for C types defined by the ISO C standard.
  - The `Storable` library, for primitive marshalling of data types between Haskell and the foreign language.

The following sections also give some hints and tips on the use of the foreign function interface in GHC.

6.5.1. 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 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 Table 7-1.

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.

6.6. 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.

6.6.1. 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 6.7).

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.

6.6.2. 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
```

6.6.3. 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 two overlapping instance declarations are permitted iff

- 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.

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*.

6.7. Explicit universal quantification

GHC now allows you to write explicitly quantified types. GHC's syntax for this now agrees with Hugs's, namely:

```
        forall a b. (Ord a, Eq  b) => a -> b -> a
```

The context is, of course, optional. You can't use `forall` as a type variable any more!

Haskell type signatures are implicitly quantified. The `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.

6.7.1. Universally-quantified data type fields

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 now have so-called *rank 2* polymorphic types, in which there is a for-all in the argument 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.)

6.7.2. Construction

You construct values of types `T1, MonadT, Swizzle` by applying the constructor to suitable values, just as usual. For example,

```
(T1 (\xy->x) 3) :: T Int

(MkSwizzle sort)    :: Swizzle
(MkSwizzle reverse) :: Swizzle

(let r x = Just x
     b m k = case m of
               Just y -> k y
               Nothing -> Nothing
  in
  MkMonad r b) :: MonadT Maybe
```

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.)

6.7.3. Pattern matching

When you use pattern matching, the bound variables may now have polymorphic types. For example:

```
f :: T a -> a -> (a, Char)
f (T1 f k) x = (f k x, f ’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.

You cannot pattern-match against an argument that is polymorphic. For example:

```
newtype TIM s a = TIM (ST s (Maybe a))

runTIM :: (forall s. TIM s a) -> Maybe a
runTIM (TIM m) = runST m
```

Here the pattern-match fails, because you can’t pattern-match against an argument of type `(forall s. TIM s a)`. Instead you must bind the variable and pattern match in the right hand side:

```
runTIM :: (forall s. TIM s a) -> Maybe a
runTIM tm = case tm of { TIM m -> runST m }
```

The `tm` on the right hand side is (invisibly) instantiated, like any polymorphic value at its occurrence site, and now you can pattern-match against it.

6.7.4. The partial-application restriction

There is really only one way in which data structures with polymorphic components might surprise you: you must not partially apply them. For example, this is illegal:

```
map MkSwizzle [sort, reverse]
```

The restriction is this: *every subexpression of the program must have a type that has no for-alls, except that in a function application (f e1. . . en) the partial applications are not subject to this rule*. The restriction makes type inference feasible.

In the illegal example, the sub-expression `MkSwizzle` has the polymorphic type `(Ord b => [b] -> [b]) -> Swizzle` and is not a sub-expression of an enclosing application. On the other hand, this expression is OK:

```
        map (T1 (\a b -> a)) [1,2,3]
```

even though it involves a partial application of `T1`, because the sub-expression `T1 (\a b -> a)` has type `Int -> T Int`.

6.7.5. Type signatures

Once you have data constructors with universally-quantified fields, or constants such as `runST` that have rank-2 types, it isn't long before you discover that you need more! Consider:

```
  mkTs f x y = [T1 f x, T1 f y]
```

`mkTs` is a fuction that constructs some values of type `T`, using some pieces passed to it. The trouble is that since `f` is a function argument, Haskell assumes that it is monomorphic, so we'll get a type error when applying `T1` to it. This is a rather silly example, but the problem really bites in practice. Lots of people trip over the fact that you can't make "wrappers functions" for `runST` for exactly the same reason. In short, it is impossible to build abstractions around functions with rank-2 types.

The solution is fairly clear. We provide the ability to give a rank-2 type signature for *ordinary* functions (not only data constructors), thus:

```
  mkTs :: (forall b. b -> b -> b) -> a -> [T a]
  mkTs f x y = [T1 f x, T1 f y]
```

This type signature tells the compiler to attribute `f` with the polymorphic type `(forall b. b -> b -> b)` when type checking the body of `mkTs`, so now the application of `T1` is fine.

There are two restrictions:

- You can only define a rank 2 type, specified by the following grammar:

  ```
  rank2type ::= [forall tyvars .] [context =>] funty
  funty     ::= ([forall tyvars .] [context =>] ty) -> funty
              | ty
  ty        ::= ...current Haskell monotype syntax...
  ```

  Informally, the universal quantification must all be right at the beginning, or at the top level of a function argument.

- There is a restriction on the definition of a function whose type signature is a rank-2 type: the polymorphic arguments must be matched on the left hand side of the "=" sign. You can't define `mkTs` like this:

```
mkTs :: (forall b. b -> b -> b) -> a -> [T a]
mkTs = \ f x y -> [T1 f x, T1 f y]
```

The same partial-application rule applies to ordinary functions with rank-2 types as applied to data constructors.

6.7.6. Type synonyms and hoisting

GHC also allows you to write a `forall` in a type synonym, thus:

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

f :: Discard a
f x y = x
```

However, it is often convenient to use these sort of 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 a. type2
==>
  forall a. 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
```

6.8. 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.

6.8.1. Why existential?

What has this to do with *existential* quantification? Simply that `MkFoo` has the (nearly) isomorphic type

```
  MkFoo :: (exists a . (a, a -> Bool)) -> Foo
```

But Haskell programmers can safely think of the ordinary *universally* quantified type given above, thereby avoiding adding a new existential quantification construct.

6.8.2. 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 (Baz1 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.

6.8.3. 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 }
```

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!

6.9. 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 `Exception.assert`, so you can still define and use your own versions of `assert`, should you so wish. If not, import `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 `Exception` library (Section 4.8 in *Haskell Libraries*) for the details.

6.10. 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`.

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.

6.10.1. Scope and implicit quantification

- All the type variables mentioned in the patterns for a single function definition equation, that are not already in scope, are brought into scope by the patterns. We describe this set as the *type variables bound by the equation*.
- The type variables thus brought into scope 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.

- There is no implicit universal quantification on pattern type signatures, nor may one write an explicit `forall` type in a pattern type signature. The pattern type signature is a monotype.
- 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).

6.10.2. Polymorphism

- Pattern type signatures are completely orthogonal to ordinary, separate type signatures. The two can be used independently or together. There is no scoping associated with the names of the type variables in a separate type signature.

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

- The function must be polymorphic in the type variables bound by all its equations. Operationally, the type variables bound by one equation must not:
  - Be unified with a type (such as `Int`, or `[a]`).
  - Be unified with a type variable free in the environment.
  - Be unified with each other. (They may unify with the type variables bound by another equation for the same function, of course.)

For example, the following all fail to type check:

```
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]
```

- The pattern-bound type variable may, however, be constrained by the context of the principal type, thus:

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

gets the inferred type: `forall a. Num a => a -> a -> a`.

6.10.3. 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.

6.10.4. Pattern signatures on other constructs

- A pattern type signature can be on an arbitrary sub-pattern, not just 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)
```

Type variables bound by these patterns must be polymorphic in the sense defined above. For example:

```
f1 (x::c) = f1 x      - ok
f2 = \(x::c) -> f2 x  - not ok
```

Here, `f1` is OK, but `f2` is not, because `c` gets unified with a type variable free in the environment, in this case, the type of `f2`, which is in the environment when the lambda abstraction is checked.

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

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

The pattern-bound type variables must, as usual, be polymorphic in the following sense: each case alternative, considered as a lambda abstraction, must be polymorphic. Thus this is OK:

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

Even though the context is that of a pair of booleans, the alternative itself is polymorphic. Of course, it is also OK to say:

```
case (True,False) of { (x::Bool, y) -> 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 that bind new type variables may not be used in pattern bindings at all. So this is illegal:

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

But these are OK, because they do not bind fresh type variables:

```
f1 x            = let (y, z::Int) = x in ...
f2 (x::(Int,a)) = let (y, z::a)   = x in ...
```

However a single variable is considered a degenerate function binding, rather than a degerate pattern binding, so this is permitted, even though it binds a type variable:

```
f :: (b->b) = \(x::b) -> x
```

Such degnerate function bindings do not fall under the monomorphism restriction. Thus:

```
g :: a -> a -> Bool = \x y. x==y
```

Here `g` has type `forall a. Eq a => a -> a -> Bool`, just as if `g` had a separate type signature. Lacking a type signature, `g` would get a monomorphic type.

6.10.5. Existentials

- 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]
  ```

6.11. Pragmas

GHC supports several pragmas, or instructions to the compiler placed in the source code. Pragmas don't affect the meaning of the program, but they might affect the efficiency of the generated code.

6.11.1. 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
```

6.11.2. 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.

6.11.3. 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 #-}
```

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

```
{-# SPECIALIZE hammeredLookup :: ...as before... = blah #-}
```

It's *Your Responsibility* to make sure that `blah` really behaves as a specialised version of `hammeredLookup`!!!

NOTE: the `=blah` feature isn't implemented in GHC 4.xx.

An example in which the `= blah` form will Win Big:

```
toDouble :: Real a => a -> Double
toDouble = fromRational . toRational

{-# SPECIALIZE toDouble :: Int -> Double = 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.

By using the US spelling, your `SPECIALIZE` pragma will work with HBC, too. Note that HBC doesn't support the `= blah` form.

A `SPECIALIZE` pragma for a function can be put anywhere its type signature could be put.

6.11.4. SPECIALIZE instance pragma

Same idea, except for instance declarations. For example:

```
instance (Eq a) => Eq (Foo a) where { ... usual stuff ... }

{-# SPECIALIZE instance Eq (Foo [(Int, Bar)] #-}
```

Compatible with HBC, by the way.

6.11.5. 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.

6.11.6. RULES pragma

The RULES pragma lets you specify rewrite rules. It is described in Section 6.12.

6.12. 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
#-}
```

6.12.1. 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 "wrong1", 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.

6.12.2. 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 6.12.4 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.

6.12.3. 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)
- `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.

6.12.4. 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).

6.12.5. 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`.

6.13. 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.

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

Again, check Simon’s Web page for publications about Parallel Haskell (including “GUM”, the key bits of the runtime system).

Some details about Parallel Haskell follow. For more information about concurrent Haskell, see Chapter 2 in *Haskell Libraries*.

6.13.1. Features specific to Parallel Haskell

6.13.1.1. 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.

6.13.1.2. 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. . .)

6.13.1.3. Scheduling policy for concurrent/parallel 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.

6.14. 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 8) for information about crashes, space leaks, and other undesirable phenomena.

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

6.14.1. 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 3.9.1).

Single quotes in module names:

It might work, but it's just begging for trouble.

6.14.2. Declarations and bindings

None known.

6.14.3. 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 will be comprehensively screwed.

6.14.4. Numbers, basic types, and built-in classes

Unchecked arithmetic:

Arguably *not* an infelicity, but... Bear in mind that operations on `Int`, `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).

Use `Integer`, `Rational`, etc., numeric types if this stuff keeps you awake at night.

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 ])
```

6.14.5. In Prelude support

Arbitrary-sized tuples:

> Plain old tuples of arbitrary size *do* work.
>
> HOWEVER: standard instances for tuples (`Eq`, `Ord`, `Bounded`, `Ix` `Read`, and `Show`) are available *only* up to 5-tuples.
>
> These limitations are easily subvertible, so please ask if you get stuck on them.

Unicode character set:

> Haskell 98 embraces the Unicode character set, but GHC doesn’t handle it. Yet.

Chapter 7. Foreign function interface

7.1. Introduction

The motivation behind this foreign function interface (FFI) specification is to make it possible to describe in Haskell *source code* the interface to foreign functionality in a Haskell system independent manner. It builds on experiences made with the previous foreign function interfaces provided by GHC and Hugs. However, the FFI specified in this document is not in the market of trying to completely bridge the gap between the actual type of an external function, and what is a *convenient* type for that function to the Haskell programmer. That is the domain of tools like HaskellDirect or Green Card, both of which are capable of generating Haskell code that uses this FFI.

Generally, the FFI consists of three parts:

1. extensions to the base language Haskell 98 (most notably `foreign import` and `foreign export` declarations), which are specified in the present document,
2. a low-level marshalling library, which is part of the *Language* part of the *Haskell Extension Library* (see Section 4.25 in *Haskell Libraries*), and a
3. a high-level marshalling library, which is still under development.

Before diving into the details of the language extension coming with the FFI, let us briefly outline the two other components of the interface.

The low-level marshalling library consists of a portion that is independent of the targeted foreign language and dedicated support for Haskell bindings to C libraries (special support for other languages may be added in the future). The language independent part is given by the module `Foreign` module (see Section 4.9 in *Haskell Libraries*). It provides support for handling references to foreign structures, for passing references to Haskell structures out to foreign routines, and for storing primitive data types in raw memory blocks in a portable manner. The support for C libraries essentially provides Haskell representations for all basic types of C (see Section 4.5 in *Haskell Libraries* and Section 4.6 in *Haskell Libraries*).

The high-level library, of which the interface definition is not yet finalised, provides routines for marshalling complex Haskell structures as well as handling out and in-out parameters in a convenient, yet protable way.

In the following, we will discuss the language extensions of the FFI (ie, the first point above). They can be split up into two complementary halves; one half that provides Haskell constructs for importing foreign functionality into Haskell, the other which lets you expose Haskell functions to the outside world. We start with the former, how to import external functionality into Haskell.

7.2. Calling foreign functions

To bind a Haskell variable name and type to an external function, we introduce a new construct:

`foreign import`. It defines the type of a Haskell function together with the name of an external function that actually implements it. The syntax of `foreign import` construct is as follows:

```
topdecl
  : ...
  ..
  | ’foreign’ ’import’ [callconv] [ext_fun] [’un-
safe’] varid ’::’ prim_type
```

A `foreign import` declaration is only allowed as a toplevel declaration. It consists of two parts, one giving the Haskell type (`prim_type`), Haskell name (`varid`) and a flag indicating whether the primitive is unsafe, the other giving details of the name of the external function (`ext_fun`) and its calling interface (`callconv`.)

Giving a Haskell name and type to an external entry point is clearly an unsafe thing to do, as the external name will in most cases be untyped. The onus is on the programmer using `foreign import` to ensure that the Haskell type given correctly maps on to the type of the external function. Section Section 7.2.5 specifies the mapping from Haskell types to external types.

7.2.1. Giving the external function a Haskell name

The external function has to be given a Haskell name. The name must be a Haskell `varid`, so the language rules regarding variable names must be followed, i.e., it must start with a lower case letter followed by a sequence of alphanumeric (‘in the Unicode sense’) characters or ’. [1]

```
varid : small ( small | large | udigit | ’ )*
```

7.2.2. Naming the external function

The name of the external function consists of two parts, one specifying its location, the other its name:

```
ext_fun  : ext_loc ext_name
         | ext_name

ext_name : string
ext_loc  : string
```

For example,

```
foreign import stdcall "Advapi32" "RegCloseKey" regCloseKey :: Addr -
> IO ()
```

states that the external function named `RegCloseKey` at location `Advapi32` should be bound to the Haskell name `regCloseKey`. For a Win32 Haskell implementation that supports the loading of DLLs on-the-fly, this declaration will most likely cause the run-time system to load the `Advapi32.dll` DLL before looking up the function `RegCloseKey()` therein to get at the function pointer to use when invoking `regCloseKey`.

Compiled implementations may do something completely different, i.e., mangle "RegCloseKey" to convert it into an archive/import library symbol, that's assumed to be in scope when linking. The details of which are platform (and compiler command-line) dependent.

If the location part is left out, the name of the external function specifies a symbol that is assumed to be in scope when linking.

The location part can either contain an absolute 'address' (i.e., path) of the archive/DLL, or just its name, leaving it up to the underlying system (system meaning both RTS/compiler and OS) to resolve the name to its real location.

An implementation is *expected* to be able to intelligently transform the `ext_loc` location to fit platform-specific practices for naming dynamic libraries. For instance, given the declaration

```
foreign import "Foo" "foo" foo :: Int -> Int -> IO ()
```

an implementation should map `Foo` to `"Foo.dll"` on a Win32 platform, and `libFoo.so` on ELF platforms. If the lookup of the dynamic library with this transformed location name should fail, the implementation should then attempt to use the original name before eventually giving up. As part of their documentation, implementations of `foreign import` should specify the exact details of how `ext_locs` are transformed and resolved, including the list of directories searched (and the order in which they are.)

In the case the Haskell name of the imported function is identical to the external name, the `ext_fun` can be omitted. i.e.,

```
foreign import sin :: Double -> IO Double
```

is identical to

```
foreign import "sin" sin :: Double -> IO Double
```

7.2.3. Calling conventions

The number of calling conventions supported is fixed:

```
callconv : ccall | stdcall
```

`ccall`

The ’default’ calling convention on a platform, i.e., the one used to do (C) function calls.

In the case of x86 platforms, the caller pushes function arguments from right to left on the C stack before calling. The caller is responsible for popping the arguments off of the C stack on return.

`stdcall`

A Win32 specific calling convention. The same as `ccall`, except that the callee cleans up the C stack before returning. [2]

Some remarks:

- Interoperating well with external code is the name of the game here, so the guiding principle when deciding on what calling conventions to include in `callconv` is that there’s a demonstrated need for a particular calling convention. Should it emerge that the inclusion of other calling conventions will generally improve the quality of this Haskell FFI, they will be considered for future inclusion in `callconv`.
- Supporting `stdcall` (and perhaps other platform-specific calling conventions) raises the issue of whether a Haskell FFI should allow the user to write platform-specific Haskell code. The calling convention is clearly an integral part of an external function’s interface, so if the one used differs from the standard one specified by the platform’s ABI *and* that convention is used by a non-trivial amount of external functions, the view of the FFI authors is that a Haskell FFI should support it.
- For `foreign import` (and other `foreign` declarations), supplying the calling convention is optional. If it isn’t supplied, it is treated as if `ccall` was specified. Users are encouraged to leave out the specification of the calling convention, if possible.

7.2.4. External function types

The range of types that can be passed as arguments to an external function is restricted (as are the range of results coming back):

```
prim_type : IO prim_result
          | prim_result
          | prim_arg '->' prim_type
```

- If you associate a non-IO type with an external function, you have the same ’proof obligations’ as when you make use of `IOExts.unsafePerformIO` in your Haskell programs.
- The external function is strict in all its arguments.
- *GHC only:* The GHC FFI implementation provides one extension to `prim_type`:

```
prim_type : ...
          | unsafe_arr_ty '->' prim_type

unsafe_arr_ty : ByteArray a
              | MutableByteArray i s a
```

GHC permits the passing of its byte array primitive types to external functions. There's some restrictions on when they can be used; see Section Section 7.2.4.1 for more details.

Section Section 7.2.4.2 defines `prim_result`; Section Section 7.2.4.1 defines `prim_arg`.

7.2.4.1. Argument types

The external function expects zero or more arguments. The set of legal argument types is restricted to the following set:

```
prim_arg : ext_ty | new_ty | ForeignObj

new_ty : a Haskell newtype of a prim_arg.

ext_ty : int_ty   | word_ty | float_ty
       | Addr     | Char    | StablePtr a
       | Bool

int_ty       : Int   | Int8   | Int16   | Int32 | Int64
word_ty      : Word8 | Word16 | Word32  | Word64
float_ty     : Float | Double
```

- `ext_ty` represent the set of basic types supported by C-like languages, although the numeric types are explicitly sized. The *stable pointer* `StablePtr` type looks out of place in this list of C-like types, but it has a well-defined and simple C mapping, see Section Section 7.2.5 for details.
- `prim_arg` represent the set of permissible argument types. In addition to `ext_ty`, `ForeignObj` is also included. The `ForeignObj` type represent values that are pointers to some external entity/object. It differs from the `Addr` type in that `ForeignObj`s are *finalized*, i.e., once the garbage collector determines that a `ForeignObj` is unreachable, it will invoke a finalising procedure attached to the `ForeignObj` to notify the outside world that we're through with using it.
- Haskell `newtype`s that wrap up a `prim_arg` type can also be passed to external functions.
- Haskell type synonyms for any of the above can also be used in `foreign import` declarations. Qualified names likewise, i.e. `Word.Word32` is legal.
- `foreign import` does not support the binding to external constants/variables. A `foreign import` declaration that takes no arguments represent a binding to a function with no arguments.
- *GHC only:* GHC's implementation of the FFI provides two extensions:

- Support for passing heap allocated byte arrays to an external function

  ```
  prim_type : ...
            | prim_arg '->' prim_type
            | unsafe_arr_ty '->' prim_type

  unsafe_arr_ty : ByteArray a
                | MutableByteArray i s a
  ```

  GHC's `ByteArray` and `MutableByteArray` primitive types are (im)mutable chunks of memory allocated on the Haskell heap, and pointers to these can be passed to `foreign imported` external functions provided they are marked as `unsafe`. Since it is inherently unsafe to hand out references to objects in the Haskell heap if the external call may cause a garbage collection to happen, you have to annotate the `foreign import` declaration with the attribute `unsafe`. By doing so, the user explicitly states that the external function won't provoke a garbage collection, so passing out heap references to the external function is allright.

- Another GHC extension is the support for unboxed types:

  ```
  prim_arg : ...  | unboxed_h_ty
  ext_ty   : .... | unboxed_ext_ty

  unboxed_ext_ty : Int#   | Word#    | Char#
                 | Float# | Double#  | Addr#
         | StablePtr# a
  unboxed_h_ty : MutableByteArray# | ForeignObj#
               | ByteArray#
  ```

  Clearly, if you want to be portable across Haskell systems, using system-specific extensions such as this is not advisable; avoid using them if you can. (Support for using unboxed types might be withdrawn sometime in the future.)

7.2.4.2. Result type

An external function is permitted to return the following range of types:

```
prim_result : ext_ty | new_ext_ty | ()

new_ext_ty : a Haskell newtype of an ext_ty.
```

where `()` represents `void` / no result.

- External functions cannot raise exceptions (IO exceptions or non-IO ones.) It is the responsibility of the `foreign import` user to layer any error handling on top of an external function.
- Only external types (`ext_ty`) can be passed back, i.e., returning `ForeignObj`s is not supported/allowed.

- Haskell newtypes that wrap up `ext_ty` are also permitted.

7.2.5. Type mapping

For the FFI to be of any practical use, the properties and sizes of the various types that can be communicated between the Haskell world and the outside, needs to be precisely defined. We do this by presenting a mapping to C, as it is commonly used and most other languages define a mapping to it. Table Table 7-1 defines the mapping between Haskell and C types.

Table 7-1. Mapping of Haskell types to C types

Haskell type	C type	requirement	range (9)
`Char`	`HsChar`	unspec. integral type	`HS_CHAR_MIN .. HS_CHAR_MAX`
`Int`	`HsInt`	signed integral of unspec. size(4)	`HS_INT_MIN .. HS_INT_MAX`
`Int8 (2)`	`HsInt8`	8 bit signed integral	`HS_INT8_MIN .. HS_INT8_MAX`
`Int16 (2)`	`HsInt16`	16 bit signed integral	`HS_INT16_MIN .. HS_INT16_MAX`
`Int32 (2)`	`HsInt32`	32 bit signed integral	`HS_INT32_MIN .. HS_INT32_MAX`
`Int64 (2,3)`	`HsInt64`	64 bit signed integral (3)	`HS_INT64_MIN .. HS_INT64_MAX`
`Word8 (2)`	`HsWord8`	8 bit unsigned integral	`0 .. HS_WORD8_MAX`
`Word16 (2)`	`HsWord16`	16 bit unsigned integral	`0 .. HS_WORD16_MAX`
`Word32 (2)`	`HsWord32`	32 bit unsigned integral	`0 .. HS_WORD32_MAX`
`Word64 (2,3)`	`HsWord64`	64 bit unsigned integral (3)	`0 .. HS_WORD64_MAX`
`Float`	`HsFloat`	floating point of unspec. size (5)	(10)
`Double`	`HsDouble`	floating point of unspec. size (5)	(10)
`Bool`	`HsBool`	unspec. integral type	(11)
`Addr`	`HsAddr`	void* (6)	
`ForeignObj`	`HsForeignObj`	void* (7)	
`StablePtr`	`HsStablePtr`	void* (8)	

Some remarks:

1. A Haskell system that implements the FFI will supply a header file `HsFFI.h` that includes target platform specific definitions for the above types and values.
2. The sized numeric types `Hs{Int,Word}{8,16,32,64}` have a 1-1 mapping to ISO C 99's `{,u}int{8,16,32,64}_t`. For systems that doesn't support this revision of ISO C, a best-fit mapping onto the supported C types is provided.
3. An implementation which does not support 64 bit integral types on the C side should implement `Hs{Int,Word}64` as a struct. In this case the bounds `HS_INT64_{MIN,MAX}` and `HS_WORD64_MAX` are undefined.
4. A valid Haskell representation of `Int` has to be equal to or wider than 30 bits. The `HsInt` synonym is guaranteed to map onto a C type that satisifies Haskell's requirement for `Int`.
5. It is guaranteed that `Hs{Float,Double}` are one of C's floating-point types `float`/`double`/`long double`.
6. It is guaranteed that `HsAddr` is of the same size as `void*`, so any other pointer type can be converted to and from HsAddr without any loss of information (K&R, Appendix A6.8).
7. Foreign objects are handled like `Addr` by the FFI, so there is again the guarantee that `HsForeignObj` is the same as `void*`. The separate name is meant as a reminder that there is a finalizer attached to the object pointed to.
8. Stable pointers are passed as addresses by the FFI, but this is only because a `void*` is used as a generic container in most APIs, not because they are real addresses. To make this special case clear, a separate C type is used here.
9. The bounds are preprocessor macros, so they can be used in `#if` and for array bounds.
10. Floating-point limits are a little bit more complicated, so preprocessor macros mirroring ISO C's `float.h` are provided:

    ```
    HS_{FLOAT,DOUBLE}_RADIX
    HS_{FLOAT,DOUBLE}_ROUNDS
    HS_{FLOAT,DOUBLE}_EPSILON
    HS_{FLOAT,DOUBLE}_DIG
    HS_{FLOAT,DOUBLE}_MANT_DIG
    HS_{FLOAT,DOUBLE}_MIN
    HS_{FLOAT,DOUBLE}_MIN_EXP
    HS_{FLOAT,DOUBLE}_MIN_10_EXP
    HS_{FLOAT,DOUBLE}_MAX
    HS_{FLOAT,DOUBLE}_MAX_EXP
    HS_{FLOAT,DOUBLE}_MAX_10_EXP
    ```

11. It is guaranteed that Haskell's `False`/`True` map to C's `0`/`1`, respectively, and vice versa. The mapping of any other integral value to `Bool` is left unspecified.
12. To avoid name clashes, identifiers starting with `Hs` and macros starting with `HS_` are reserved for the FFI.

13. *GHC only:* The GHC specific types `ByteArray` and `MutableByteArray` both map to `char*`.

7.2.6. Some `foreign import` wrinkles

- By default, a `foreign import` function is *safe*. A safe external function may cause a Haskell garbage collection as a result of being called. This will typically happen when the imported function end up calling Haskell functions that reside in the same 'Haskell world' (i.e., shares the same storage manager heap) – see Section Section 7.4 for details of how the FFI let's you call Haskell functions from the outside. If the programmer can guarantee that the imported function won't call back into Haskell, the `foreign import` can be marked as 'unsafe' (see Section Section 7.2 for details of how to do this.) Unsafe calls are cheaper than safe ones, so distinguishing the two classes of external calls may be worth your while if you're extra conscious about performance.
- A `foreign import`ed function should clearly not need to know that it is being called from Haskell. One consequence of this is that the lifetimes of the arguments that are passed from Haskell *must* equal that of a normal C call. For instance, for the following decl,

```
foreign import "mumble" mumble :: ForeignObj -> IO ()

f :: Addr -> IO ()
f ptr = do
  fo <- newForeignObj ptr myFinalizer
  mumble fo
```

The `ForeignObj` must live across the call to `mumble` even if it is not subsequently used/reachable. Why the insistence on this? Consider what happens if `mumble` calls a function which calls back into the Haskell world to execute a function, behind our back as it were. This evaluation may possibly cause a garbage collection, with the result that `fo` may end up being finalised. By guaranteeing that `fo` will be considered live across the call to `mumble`, the unfortunate situation where `fo` is finalised (and hence the reference passed to `mumble` is suddenly no longer valid) is avoided.

7.3. Invoking external functions via a pointer

A `foreign import` declaration imports an external function into Haskell. (The name of the external function is statically known, but the loading/linking of it may very well be delayed until run-time.) A `foreign import` declaration is then (approximately) just a type cast of an external function with a *statically known name*.

An extension of `foreign import` is the support for *dynamic* type casts of external names/addresses:

```
topdecl
   : ...
   ..
   | 'foreign' 'import' [callconv] 'dynamic' ['unsafe']
            varid :: Addr -> (prim_args -> IO prim_result)
```

i.e., identical to a `foreign import` declaration, but for the specification of `dynamic` instead of the name of an external function. The presence of `dynamic` indicates that when an application of `varid` is evaluated, the function pointed to by its first argument will be invoked, passing it the rest of `varid`'s arguments.

What are the uses of this? Native invocation of COM methods, [3] Haskell libraries that want to be dressed up as C libs (and hence may have to support C callbacks), Haskell code that need to dynamically load and execute code.

7.4. Exposing Haskell functions

So far we've provided the Haskell programmer with ways of importing external functions into the Haskell world. The other half of the FFI coin is how to expose Haskell functionality to the outside world. So, dual to the `foreign import` declaration is `foreign export`:

```
topdecl
  : ...
  ..
  | 'foreign' 'export' callconv [ext_name] varid :: prim_type
```

A `foreign export` declaration tells the compiler to expose a locally defined Haskell function to the outside world, i.e., wrap it up behind a calling interface that's useable from C. It is only permitted at the toplevel, where you have to specify the type at which you want to export the function, along with the calling convention to use. For instance, the following export declaration:

```
foreign export ccall "foo" bar :: Int -> Addr -> IO Double
```

will cause a Haskell system to generate the following C callable function:

```
HsDouble foo(HsInt arg1, HsAddr arg2);
```

When invoked, it will call the Haskell function `bar`, passing it the two arguments that was passed to `foo()`.

- The range of types that can be passed as arguments and results is restricted, since `varid` has got a `prim_type`.
- It is not possible to directly export operator symbols.

- The type checker will verify that the type given for the `foreign export` declaration is compatible with the type given to function definition itself. The type in the `foreign export` may be less general than that of the function itself. For example, this is legal:

  ```
  f :: Num a => a -> a
  foreign export ccall "fInt"   f :: Int -> Int
  foreign export ccall "fFloat" f :: Float -> Float
  ```

  These declarations export two C-callable procedures `fInt` and `fFloat`, both of which are implemented by the (overloaded) Haskell function `f`.

- The `foreign exported` IO action must catch all exceptions, as the FFI does not address how to signal Haskell exceptions to the outside world.

7.4.1. Exposing Haskell function values

The `foreign export` declaration gives the C programmer access to statically defined Haskell functions. It does not allow you to conveniently expose dynamically-created Haskell function values as C function pointers though. To permit this, the FFI supports *dynamic* `foreign exports`:

```
topdecl
  : ...
  ..
  | ’foreign’ ’export’ [callconv] ’dynamic’ varid :: prim_type -> IO Addr
```

A `foreign export dynamic` declaration declares a C function pointer *generator*. Given a Haskell function value of some restricted type, the generator wraps it up behind an externally callable interface, returning an `Addr` to an externally callable (C) function pointer.

When that function pointer is eventually called, the corresponding Haskell function value is applied to the function pointer’s arguments and evaluated, returning the result (if any) back to the caller.

The mapping between the argument to a `foreign export dynamic` declaration and its corresponding C function pointer type, is as follows:

```
typedef cType[[Res]] (*Varid_FunPtr)
        (cType[[Ty_1]] ,.., cType[[Ty_n]]);
```

where `cType[[]]` is the Haskell to C type mapping presented in Section Section 7.2.5.

To make it all a bit more concrete, here’s an example:

```
foreign export dynamic mkCallback :: (Int -> IO Int) -> IO Addr

foreign import registerCallback :: Addr -> IO ()

exportCallback :: (Int -> IO Int) -> IO ()
exportCallback f = do
```

```
  fx <- mkCallback f
  registerCallback fx
```

The `exportCallback` lets you register a Haskell function value as a callback function to some external library. The C type of the callback that the external library expects in `registerCallback()`, is: [4]

```
typedef HsInt (*mkCallback_FunPtr) (HsInt arg1);
```

Creating the view of a Haskell closure as a C function pointer entails registering the Haskell closure as a ’root’ with the underlying Haskell storage system, so that it won’t be garbage collected. The FFI implementation takes care of this, but when the outside world is through with using a C function pointer generated by a `foreign export dynamic` declaration, it needs to be explicitly freed. This is done by calling:

```
void freeHaskellFunctionPtr(void *ptr);
```

In the event you need to free these function pointers from within Haskell, a standard ’foreign import’ed binding of the above C entry point is also provided,

```
Foreign.freeHaskellFunctionPtr :: Addr -> IO ()
```

7.4.2. Code addresses

The `foreign import` declaration allows us to invoke an external function by name from within the comforts of the Haskell world, while `foreign import dynamic` lets us invoke an external function by address. However, there’s no way of getting at the code address of some particular external label though, which is at times useful, e.g. for the construction of method tables for, say, Haskell COM components. To support this, the FFI has got `foreign labels`:

```
foreign label "freeAtLast" addrOf_freeAtLast :: Addr
```

The meaning of this declaration is that `addrOf_freeAtLast` will now contain the address of the label `freeAtLast`.

Notes

1. Notice that with Haskell 98, underscore (’_’) is included in the character class `small`.
2. The `stdcall` is a Microsoft Win32 specific wrinkle; it used throughout the Win32 API, for instance. On platforms where `stdcall` isn’t meaningful, it should be treated as being equal to

`ccall`.

3. Or the interfacing to any other software component technologies.
4. An FFI implementation is encouraged to generate the C typedef corresponding to a `foreign export dynamic` declaration, but isn't required to do so.

Chapter 8. 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 8.3 for 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 6.14, which describes Glasgow Haskell's shortcomings vs. the Haskell language definition, may also be of interest.)

8.1. 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.

"The compiler ran out of heap (or stack) when compiling itself!"

It happens. We try to supply reasonable `-H<n>` flags for `ghc/compiler/` and `ghc/lib/`, but GHC's memory consumption can vary by platform (e.g., on a 64-bit machine).

Just say **make all EXTRA_HC_OPTS=-H<a reasonable number>** and see how you get along.

Note that this is less likely to happen if you are compiling with GHC 4.00 or later, since the introduction of the dynamically expanding heap.

"The compiler died with a pattern-matching error."

This is a bug just as surely as a "panic." Please report it.

"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 these 'trace' messages from GHC?"

Almost surely not a problem. About some specific cases...

Simplifier still going after N iterations:

Sad, but harmless. You can change the number with a `-fmax-simplifier-iterations<N>` option (no space); and you can see what actions took place in each iteration by turning on the `-fshow-simplifier-progress` option.

If the simplifier definitely seems to be "looping," please report it.

"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.

"What's a 'consistency error'?"

(These are reported just after linking your program.)

You tried to link incompatible object files, e.g., normal ones (registerised, Appel garbage-collector) with profiling ones (two-space collector). Or those compiled by a previous version of GHC with an incompatible newer version.

If you run **nm -o *.o | egrep 't (cc|hsc)\.'** (or, on unregisterised files: **what *.o**), you'll see all the consistency tags/strings in your object files. They must all be the same! (ToDo: tell you what they mean...)

"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.

8.2. When your program "does the wrong thing"

(For advice about overly slow or memory-hungry Haskell programs, please see Chapter 5).

“Help! My program crashed!”

(e.g., a ‘segmentation fault’ or ‘core dumped’)

If your program has no foreign calls in it, 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 3.7.6.

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.

If you are interested in hard-core debugging of a crashing GHC-compiled program, please see Section 8.4.

“My program entered an ‘absent’ argument.”

This is definitely caused by a bug in GHC. Please report it.

“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 architecture). Divide-by-zero *may* cause an untrapped exception (please report it if it does).

8.3. How to report a bug in the GHC system

Glasgow Haskell is a changing system so there are sure to be bugs in it. Please report them to <`glasgow-haskell-bugs@haskell.org`>! (However, please check the earlier part of this section to be sure it's not a known not-really-a problem.)

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? (**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.

8.4. Hard-core debugging of GHC-compiled programs

If your program is crashing, you should almost surely file a bug report, as outlined in previous sections.

This section suggests ways to Make Further Progress Anyway.

The first thing to establish is: Is it a garbage-collection (GC) bug? Try your program with a very large heap and a `-Sstderr` RTS flag.

- If it crashes *without* garbage-collecting, then it is definitely *not* a GC bug.
- If you can make it crash with one heap size but not with another, then it *probably is* a GC bug.
- If it crashes with the normal collector, but not when you force two-space collection (`-G1` runtime flag), then it *probably is* a GC bug.

If it *is* a GC bug, you may be able to avoid it by using a particular heap size or by using a `-G1` runtime flag. (But don't forget to report the bug!!!)

ToDo: more here?

Chapter 9. Other Haskell utility programs

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

9.1. Emacs 'TAGS' for Haskell: hstags

'Tags' is a facility for indexing the definitions of programming-language things in a multi-file program, and then using that index to jump around among these definitions.

Rather than scratch your head, saying "Now where did we define 'foo'?", you just do (in Emacs) `M-. foo RET`, and You're There! Some people go wild over this stuff…

GHC comes with a program **hstags**, which build Emacs-able TAGS files. The invocation syntax is:

```
hstags [GHC-options] file [files...]
```

The best thing is just to feed it your GHC command-line flags. A good Makefile entry might be:

```
tags:
        $(RM) TAGS
        hstags $(GHC_FLAGS) *.lhs
```

The only flags of its own are: `-v` to be verbose; `-a` to *APPEND* to the TAGS file, rather than write to it.

Shortcomings: (1) Instance declarations don't get into the TAGS file (but the definitions inside them do); as instances aren't named, this is probably just as well. (2) Data-constructor definitions don't get in. Go for the corresponding type constructor instead.

(Actually, GHC also comes with **etags** [for C], and **perltags** [for You Know What]. And—I cannot tell a lie—there is Denis Howe's **fptags** [for Haskell, etc.] in the `ghc/CONTRIB` section…)

9.2. "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.

9.3. Pretty-printing Haskell: pphs

Andrew Preece has written **pphs**, a utility to pretty-print Haskell code in LaTeX documents. Keywords in bolds, variables in italics—that sort of thing. It is good at lining up program clauses and equals signs, things that are very tiresome to do by hand.

The code is distributed with GHC in `ghc/CONTRIB/pphs`.

Chapter 10. 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).

10.1. 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? :-)

10.2. 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.

10.3. Creating a DLL

Sealing up your Haskell library inside a DLL is straightforward; compile up the object files that make up the library, and then build the DLL by issuing a command of the form:

```
ghc -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 3.7.4.1) 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 10.4). 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.

10.4. 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(__init_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, __init_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:

```
gcc -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.

Haskell Libraries

The GHC Team

Haskell Libraries

by The GHC Team

Table of Contents

Chapter 1. Introduction

Hugs and GHC provide a common set of libraries to aid portability. This document specifies the interfaces to these libraries and documents known differences. It is the hope of the GHC team that these libraries in the long run become part of every Haskell system.

1.1. Naming Conventions

The set of interfaces specified in this document try to adhere to the following naming conventions:

- Actions that create a new values have the prefix `new` followed by the name of the type of object they're creating, e.g., `newIORef`, `newChan` etc.
- Operations that read a value from a mutable object are prefixed with `read`, and operations that update the contents have the prefix `write`, e.g., `readChan`, `readIOArray`. Notes:
  - This differs from the convention used to name the operations for reading and writing to a file `Handle`, where `get` and `put` are used instead.
  - Operations provided by various concurrency abstractions, e.g., `MVar`, `CVar` , also deviate from this naming scheme. This is perhaps defensible, since the read and write operations have additional behaviour, e.g., `takeMVar` tries to read the current value of an `MVar`, locking it if it succeeds.
- Conversions operators have the form `AToB` where `A` and `B` are the types we're converting between.
- Operations that lazily read values from a mutable object/handle, have the form `getXContents`, e.g., `Channel.getChanContents` and `IO.hGetContents`. (OK, so the latter isn't called `getHandleContents`, but you hopefully get the picture.)

Chapter 2. The `concurrent` category: concurrency support

2.1. Concurrent Haskell

GHC and Hugs both provide concurrency extensions, as described in Concurrent Haskell (http://www.haskell.org/ghc/docs/papers/concurrent-haskell.ps.gz).

Concurrency in GHC and Hugs is "lightweight", which means that both thread creation and context switching overheads are extremely low. Scheduling of Haskell threads is done internally in the Haskell runtime system, and doesn’t make use of any operating system-supplied thread packages.

Haskell threads can communicate via `MVars`, a kind of synchronised mutable variable (see Section 2.4.3). Several common concurrency abstractions can be built from `MVars`, and these are provided by the `Concurrent` library, which is described in the later sections. Threads may also communicate via exceptions.

2.2. Concurrency Basics

To gain access to the concurrency primitives, just `import Concurrent` in your Haskell module. In GHC, you also need to add the `-syslib concurrent` option to the command line.

To create a new thread, use `forkIO`:

```
forkIO :: IO () -> IO ThreadId
```

This sparks off a new thread to run the `IO` computation passed as the first argument.

The returned `ThreadId` is an abstract type representing a handle to the newly created thread. The `ThreadId` type is an instance of both `Eq` and `Ord`, where the `Ord` instance implements an arbitrary total ordering over `ThreadIds`.

Threads may also be killed via the `ThreadId`:

```
killThread :: ThreadId -> IO ()
```

this terminates the given thread (Note: `killThread` is not implemented in Hugs yet). Any work already done by the thread isn’t lost: the computation is suspended until required by another thread. The memory used by the thread will be garbage collected if it isn’t referenced from anywhere else.

More generally, an arbitrary exception (see Section 4.8) may be raised in any thread for which we have a `ThreadId`, with `raiseInThread`:

```
raiseInThread :: ThreadId -> Exception -> IO ()
```

Actually `killThread` just raises the `ThreadKilled` exception in the target thread, the normal action of which is to just terminate the thread. The target thread will stop whatever it was doing (even if it was blocked on an `MVar` or other computation) and handle the exception.

One important property of `raiseInThread` (and therefore `killThread`) is that they are *synchronous*, in the sense that after performing a `raiseInThread` operation, the calling thread can be certain that the target thread has received the exception. In other words, the target thread cannot perform any more processing unless it handles the exception that has just been raised in it. This is a useful property to know when dealing with race conditions: eg. if there are two threads that can kill each other, it is guaranteed that only one of the threads will get to kill the other.

The `ThreadId` for the current thread can be obtained with `myThreadId`:

```
myThreadId :: IO ThreadId
```

NOTE: if you have a `ThreadId`, you essentially have a pointer to the thread itself. This means the thread itself can't be garbage collected until you drop the `ThreadId`. This misfeature will hopefully be corrected at a later date.

2.3. Scheduling

GHC uses *preemptive multitasking*: context switches can occur at any time. At present, Hugs uses *cooperative multitasking*: context switches only occur when you use one of the primitives defined in this module. This means that programs such as:

```
main = forkIO (write ’a’) » write ’b’
 where write c = putChar c » write c
```

will print either `aaaaaaaaaaaaaa...` or `bbbbbbbbbbbb...`, instead of some random interleaving of `a`s and `b`s. In practice, cooperative multitasking is sufficient for writing simple graphical user interfaces.

The `yield` action forces a context-switch to any other currently runnable threads (if any), and is occasionally useful when implementing concurrency abstractions:

```
yield :: IO ()
```

2.3.1. Thread Waiting

There are operations to delay a concurrent thread, and to make one wait:

```
threadDelay     :: Int -> IO ()
threadWaitRead  :: Int -> IO ()
threadWaitWrite :: Int -> IO ()
```

The `threadDelay` operation will cause the current thread to suspend for a given number of microseconds. Note that the resolution used by the Haskell runtime system's internal timer together with the fact that the thread may take some time to be rescheduled after the time has expired, means that the accuracy is more like 1/50 second.

`threadWaitRead` and `threadWaitWrite` can be used to block a thread until I/O is available on a given file descriptor. These primitives are used by the I/O subsystem to ensure that a thread waiting on I/O doesn't hang the entire system.

2.3.2. Blocking

Calling a foreign C procedure (such as `getchar`) that blocks waiting for input will block *all* threads, in both GHC and Hugs. The GHC I/O system uses non-blocking I/O internally to implement thread-friendly I/O, so calling standard Haskell I/O functions blocks only the thead making the call.

2.4. Concurrency abstractions

2.4.1. `Chan`: Channels

A `Channel` is an unbounded channel:

```
data Chan a
newChan         :: IO (Chan a)
writeChan       :: Chan a -> a -> IO ()
readChan        :: Chan a -> IO a
dupChan         :: Chan a -> IO (Chan a)
unGetChan       :: Chan a -> a -> IO ()
getChanContents :: Chan a -> IO [a]
writeList2Chan :: Chan a -> [a] -> IO ()
```

2.4.2. `CVar`: Channel variables

A *channel variable* (`CVar`) is a one-element channel, as described in the paper:

```
data CVar a
newCVar   :: IO (CVar a)
writeCVar :: CVar a -> a -> IO ()
readCVar  :: CVar a -> IO a
```

2.4.3. `MVar`: Synchronising variables

The `MVar` interface provides access to "MVars" (pronounced "em-vars"), which are *synchronising variables*. An `MVar` is simply a box, which may be empty or full. The basic operations available over `MVar`s are given below:

```
data MVar a - abstract
instance Eq (MVar a)

newEmptyMVar     :: IO (MVar a)
newMVar          :: a -> IO (MVar a)
takeMVar         :: MVar a -> IO a
putMVar          :: MVar a -> a -> IO ()
readMVar         :: MVar a -> IO a
swapMVar         :: MVar a -> a -> IO a
tryTakeMVar      :: MVar a -> IO (Maybe a)
isEmptyMVar      :: MVar a -> IO Bool
```

`newEmptyMVar`
`newMVar`

New empty `MVar`s can be created with `newEmptyMVar`. To create an `MVar` with an initial value, use `newMVar`.

`takeMVar`

The `takeMVar` operation returns the contents of the `MVar` if it was full, or waits until it becomes full otherwise.

`putMVar`

The `putMVar` operation puts a value into an empty `MVar`. Calling `putMVar` on an already full `MVar` results in a `PutFullMVar` exception being raised (see Section 4.8).

`tryTakeMVar`

The `tryTakeMVar` is a non-blocking version of `takeMVar`. If the `MVar` is full, then it returns `Just a` (where `a` is the contents of the `MVar`) and empties the `MVar`. If the `MVar` is empty, it immediately returns `Nothing`.

`isEmptyMVar`

The operation `isEmptyMVar` returns a flag indicating whether the `MVar` is currently empty or filled in, i.e., will a thread block when performing a `takeMVar` on that `MVar` or not?

Please notice that the Boolean value returned from `isEmptyMVar` represent just a snapshot of the state of the `MVar`. By the time a thread gets to inspect the result and act upon it, other threads may have accessed the `MVar` and changed the 'filled-in' status of the variable. The same proviso applies to `isEmptyChan` (next sub-section).

`readMVar`

This is a combination of `takeMVar` and `putMVar`; ie. it takes the value from the `MVar`, puts it back, and also returns it.

`swapMVar`

`swapMVar` swaps the contents of an `MVar` for a new value.

2.4.4. `QSem`: General semaphores

```
data QSem
newQSem     :: Int   -> IO QSem
waitQSem    :: QSem  -> IO ()
signalQSem  :: QSem  -> IO ()
```

2.4.5. `QSemN`: Quantity semaphores

```
data QSemN
newQSemN    :: Int   -> IO QSemN
signalQSemN :: QSemN -> Int -> IO ()
waitQSemN   :: QSemN -> Int -> IO ()
```

2.4.6. `SampleVar`: Sample variables

A *sample variable* (`SampleVar`) is slightly different from a normal `MVar`:

- Reading an empty `SampleVar` causes the reader to block (same as `takeMVar` on empty `MVar`).
- Reading a filled `SampleVar` empties it and returns value. (same as `takeMVar`)
- Writing to an empty `SampleVar` fills it with a value, and potentially, wakes up a blocked reader (same as for `putMVar` on empty `MVar`).
- Writing to a filled `SampleVar` overwrites the current value. (different from `putMVar` on full `MVar`.)

```
type SampleVar a = MVar (Int, MVar a)

emptySampleVar :: SampleVar a -> IO ()
newSampleVar   :: IO (SampleVar a)
readSample     :: SampleVar a -> IO a
writeSample    :: SampleVar a -> a -> IO ()
```

2.4.7. Merging Streams

Merging streams—binary and n-ary:

```
mergeIO  :: [a]   -> [a] -> IO [a]
nmergeIO :: [[a]] -> IO [a]
```

These actions fork one thread for each input list that concurrently evaluates that list; the results are merged into a single output list.

Note: Hugs does not provide the functions `mergeIO` or `nmergeIO` since these require preemptive multitasking.

2.5. The `Concurrent` library interface

The full interface for the `Concurrent` library is given below for reference:

```
data ThreadId    - thread identifiers
instance Eq  ThreadId
instance Ord ThreadId

forkIO           :: IO () -> IO ThreadId
myThreadId       :: IO ThreadId
killThread       :: ThreadId -> IO ()
par              :: a -> b -> b
seq              :: a -> b -> b
fork             :: a -> b -> b
yield            :: IO ()

threadDelay      :: Int -> IO ()
threadWaitRead   :: Int -> IO ()
threadWaitWrite  :: Int -> IO ()

mergeIO          :: [a]   -> [a] -> IO [a]
nmergeIO  :: [[a]] ->y IO [a]

module Chan
module CVar
module MVar
module QSem
module QSemN
module SampleVar
```

2.6. GHC-specific concurrency issues

In a standalone GHC program, only the main thread is required to terminate in order for the process to terminate. Thus all other forked threads will simply terminate at the same time as the main thread (the terminology for this kind of behaviour is “daemonic threads”).

If you want the program to wait for child threads to finish before exiting, you need to program this yourself. A simple mechanism is to have each child thread write to an `MVar` when it completes, and have the main thread wait on all the `MVars` before exiting:

```
myForkIO :: IO () -> IO (MVar ())
myForkIO io = do
  mvar <- newEmptyMVar
  forkIO (io ‘finally‘ putMVar mvar ())
  return mvar
```

Note that we use `finally` from the `Exception` module to make sure that the `MVar` is written to even if the thread dies or is killed for some reason.

A better method is to keep a global list of all child threads which we should wait for at the end of the program:

```
children :: MVar [MVar ()]
children = unsafePerformIO (newMVar [])

waitForChildren :: IO ()
waitForChildren = do
  (mvar:mvars) <- takeMVar children
  putMVar children mvars
  takeMVar mvar
  waitForChildren

forkChild :: IO () -> IO ()
forkChild io = do
   mvar <- newEmptyMVar
   forkIO (p ‘finally‘ putMVar mvar ())
   childs <- takeMVar children
   putMVar children (mvar:childs)

later = flip finally

main =
  later waitForChildren $
  ...
```

Chapter 3. The `data` category: datatypes

3.1. Edison

Edison is a complete package of data structures for Haskell. Documentation can currently be found at The Edison home page (http://www.cs.columbia.edu/~cdo/edison/).

3.2. The `FiniteMap` type

What functional programmers call a *finite map*, everyone else calls a *lookup table*.

Out code is derived from that in this paper: “S Adams "Efficient sets: a balancing act" Journal of functional programming 3(4) Oct 1993, pages 553-562” Guess what? The implementation uses balanced trees.

```
data FiniteMap key elt  - abstract

-      BUILDING
emptyFM         :: FiniteMap key elt
unitFM          :: key -> elt -> FiniteMap key elt
listToFM        :: Ord key => [(key,elt)] -> FiniteMap key elt
                        - In the case of duplicates, the last is taken

-      ADDING AND DELETING
                   - Throws away any previous binding
                   -
In the list case, the items are added starting with the
                   - first one in the list
addToFM         :: Ord key => FiniteMap key elt -> key -> elt  -
> FiniteMap key elt
addListToFM     :: Ord key => FiniteMap key elt -> [(key,elt)] -
> FiniteMap key elt

                 - Combines with previous binding
                 - In the combining function, the first argument is
                 - the "old" element, while the second is the "new" one.
addToFM_C       :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> key -> elt
                           -> FiniteMap key elt
addListToFM_C   :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> [(key,elt)]
                           -> FiniteMap key elt

                 -
Deletion doesn’t complain if you try to delete something
```

```
                 - which isn’t there
delFromFM       :: Ord key => FiniteMap key elt -> key   -
> FiniteMap key elt
delListFromFM   :: Ord key => FiniteMap key elt -> [key] -
> FiniteMap key elt

-      COMBINING
                 - Bindings in right argument shadow those in the left
plusFM          :: Ord key => FiniteMap key elt -> FiniteMap key elt
                           -> FiniteMap key elt

                   -
Combines bindings for the same thing with the given function
plusFM_C        :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> FiniteMap key elt -
> FiniteMap key elt

minusFM         :: Ord key => FiniteMap key elt -> FiniteMap key elt -
> FiniteMap key elt
                   -
(minusFM a1 a2) deletes from a1 any bindings which are bound in a2

intersectFM     :: Ord key => FiniteMap key elt -> FiniteMap key elt -
> FiniteMap key elt
intersectFM_C   :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> FiniteMap key elt -
> FiniteMap key elt

-      MAPPING, FOLDING, FILTERING
foldFM          :: (key -> elt -> a -> a) -> a -> FiniteMap key elt -> a
mapFM           :: (key -> elt1 -> elt2) -> FiniteMap key elt1 -
> FiniteMap key elt2
filterFM        :: Ord key => (key -> elt -> Bool)
                           -> FiniteMap key elt -> FiniteMap key elt

-      INTERROGATING
sizeFM          :: FiniteMap key elt -> Int
isEmptyFM       :: FiniteMap key elt -> Bool

elemFM          :: Ord key => key -> FiniteMap key elt -> Bool
lookupFM        :: Ord key => FiniteMap key elt -> key -> Maybe elt
lookupWithDefaultFM
                :: Ord key => FiniteMap key elt -> elt -> key -> elt
                - lookupWithDefaultFM supplies a "default" elt
                - to return for an unmapped key

-      LISTIFYING
fmToList        :: FiniteMap key elt -> [(key,elt)]
keysFM          :: FiniteMap key elt -> [key]
```

```
eltsFM          :: FiniteMap key elt -> [elt]
```

3.3. The `Set` type

Our implementation of *sets* (key property: no duplicates) is just a variant of the `FiniteMap` module.

```
data Set        - abstract
                - instance of: Eq

emptySet        :: Set a
mkSet           :: Ord a => [a]  -> Set a
setToList       :: Set a -> [a]
unitSet         :: a -> Set a
singletonSet    :: a -> Set a  - deprecated, use unitSet.

union           :: Ord a => Set a -> Set a -> Set a
unionManySets   :: Ord a => [Set a] -> Set a
minusSet        :: Ord a => Set a -> Set a -> Set a
mapSet          :: Ord a => (b -> a) -> Set b -> Set a
intersect       :: Ord a => Set a -> Set a -> Set a

elementOf       :: Ord a => a -> Set a -> Bool
isEmptySet      :: Set a -> Bool

cardinality     :: Set a -> Int
```

Chapter 4. The `lang` category: language support

4.1. `Addr`

This library provides machine addresses, i.e., handles to chunks of raw memory. It is primarily intended for use with the Foreign Function Interface (FFI) and will usually be imported via the module `Foreign` (see Section 4.9).

4.1.1. Address Type and Arithmetic

```
data Addr      - abstract handle for memory addresses
               - instance of: Eq, Ord, Show, Typeable

data AddrOff   - abstract handle of address offsets
               -
instance of: Eq, Ord, Show, Enum, Num, Real, Integral, Typeable

nullAddr  :: Addr
alignAddr :: Addr -> Int     -> Addr
plusAddr  :: Addr -> AddrOff -> Addr
minusAddr :: Addr -> Addr    -> AddrOff
```

The following specifies the behaviour of the four function definitions.

`nullAddr :: Addr`

The constant `nullAddr` contains a distinguished value of `Addr` that denotes the absence of an address that is associated with a valid memory location.

`alignAddr :: Addr -> Int -> Addr`

Given an arbitrary address and an alignment constraint, `alignAddr` yields the next higher address that fulfills the alignment constraint. An alignment constraint `x` is fulfilled by any address divisible by `x`. This operation is idempotent.

`plusAddr :: Addr -> AddrOff -> Addr`

Advances the given address by the given address offset.

`minusAddr :: Addr -> Addr -> AddrOff`

Computes the offset required to get from the first to the second argument. We have

```
a2 == a1 ‘plusAddr‘ (a2 ‘minusAddr‘ a1)
```

4.1.2. The Standard C-side Interface

The following definition is available to C programs inter-operating with Haskell code when including the header `HsFFI.h`.

```
typedef void* HsAddr;  /* C representation of an Addr */
```

4.1.3. Deprecated Functions

The following functions are deprecated in the new FFI. Use the module `Storable` (Section 4.25) instead.

```
- read value out of _immutable_ memory
indexCharOffAddr       :: Addr -> Int -> Char
indexIntOffAddr        :: Addr -> Int -> Int
indexAddrOffAddr       :: Addr -> Int -> Addr
indexFloatOffAddr      :: Addr -> Int -> Float
indexDoubleOffAddr     :: Addr -> Int -> Double
indexWord8OffAddr      :: Addr -> Int -> Word8
indexWord16OffAddr     :: Addr -> Int -> Word16
indexWord32OffAddr     :: Addr -> Int -> Word32
indexWord64OffAddr     :: Addr -> Int -> Word64
indexInt8OffAddr       :: Addr -> Int -> Int8
indexInt16OffAddr      :: Addr -> Int -> Int16
indexInt32OffAddr      :: Addr -> Int -> Int32
indexInt64OffAddr      :: Addr -> Int -> Int64
indexStablePtrOffAddr  :: Addr -> Int -> StablePtr a

- read value out of mutable memory
readCharOffAddr        :: Addr -> Int -> IO Char
readIntOffAddr         :: Addr -> Int -> IO Int
readAddrOffAddr        :: Addr -> Int -> IO Addr
readFloatOffAddr       :: Addr -> Int -> IO Float
readDoubleOffAddr      :: Addr -> Int -> IO Double
readWord8OffAddr       :: Addr -> Int -> IO Word8
readWord16OffAddr      :: Addr -> Int -> IO Word16
readWord32OffAddr      :: Addr -> Int -> IO Word32
readWord64OffAddr      :: Addr -> Int -> IO Word64
readInt8OffAddr        :: Addr -> Int -> IO Int8
readInt16OffAddr       :: Addr -> Int -> IO Int16
readInt32OffAddr       :: Addr -> Int -> IO Int32
readInt64OffAddr       :: Addr -> Int -> IO Int64
readStablePtrOffAddr   :: Addr -> Int -> IO (StablePtr a)

- write value into mutable memory
writeCharOffAddr       :: Addr -> Int -> Char   -> IO ()
writeIntOffAddr        :: Addr -> Int -> Int    -> IO ()
writeAddrOffAddr       :: Addr -> Int -> Addr   -> IO ()
```

```
writeFloatOffAddr      :: Addr -> Int -> Float  -> IO ()
writeDoubleOffAddr     :: Addr -> Int -> Double -> IO ()
writeWord8OffAddr      :: Addr -> Int -> Word8  -> IO ()
writeWord16OffAddr     :: Addr -> Int -> Word16 -> IO ()
writeWord32OffAddr     :: Addr -> Int -> Word32 -> IO ()
writeWord64OffAddr     :: Addr -> Int -> Word64 -> IO ()
writeInt8OffAddr       :: Addr -> Int -> Int8   -> IO ()
writeInt16OffAddr      :: Addr -> Int -> Int16  -> IO ()
writeInt32OffAddr      :: Addr -> Int -> Int32  -> IO ()
writeInt64OffAddr      :: Addr -> Int -> Int64  -> IO ()
writeForeignObjOffAddr :: Addr -> Int -> ForeignObj -> IO ()
writeStablePtrOffAddr  :: Addr -> Int -> StablePtr a -> IO ()

- conversion to/from Int, a little bit doubtful...
addrToInt              :: Addr -> Int
intToAddr              :: Int  -> Addr

- completely deprecated
data Word = W# Word#
wordToInt              :: Word -> Int
intToWord              :: Int  -> Word
```

4.1.4. Hugs Specifics

Hugs provides `Addr` and `nullAddr` but does not provide any of the index, read or write functions. They can be implemented using GreenCard if required.

4.2. `Bits`

This module defines bitwise operations for signed and unsigned ints. Instances of `class Bits` can be obtained from the `Int` (Section 4.13) and `Word` (Section 4.27) modules.

```
infixl 8 ‘shift‘, ‘rotate‘
infixl 7 .&.
infixl 6 ‘xor‘
infixl 5 .|.

class Bits a where
  (.&.), (.|.), xor :: a -> a -> a
  complement        :: a -> a
  shift             :: a -> Int -> a
  rotate            :: a -> Int -> a
  bit               :: Int -> a
  setBit            :: a -> Int -> a
  clearBit          :: a -> Int -> a
  complementBit     :: a -> Int -> a
  testBit           :: a -> Int -> Bool
  bitSize           :: a -> Int
  isSigned          :: a -> Bool

shiftL, shiftR   :: Bits a => a -> Int -> a
rotateL, rotateR :: Bits a => a -> Int -> a
shiftL  a i = shift  a i
shiftR  a i = shift  a (-i)
rotateL a i = rotate a i
rotateR a i = rotate a (-i)
```

Notes:

- `bitSize` and `isSigned` are like `floatRadix` and `floatDigits`—they return parameters of the *type* of their argument rather than of the particular argument they are applied to. `bitSize` returns the number of bits in the type; and `isSigned` returns whether the type is signed or not.
- `shift` performs sign extension on signed number types. That is, right shifts fill the top bits with 1 if the number is negative and with 0 otherwise.
- Bits are numbered from 0 with bit 0 being the least significant bit.
- `shift x i` and `rotate x i` shift to the left if `i` is positive and to the right otherwise.
- `bit i` is the value with the i'th bit set.

4.3. `ByteArray`

NOTE: The `ByteArray` interface is deprecated, please use `IArray` (Section 4.12) or `MArray` (Section 4.16) instead.

`ByteArray`s are chunks of immutable Haskell heap:

```
data ByteArray ix - abstract

instance Eq (ByteArray ix)

newByteArray        :: Ix ix => (ix,ix) -> ST s (ByteArray ix)

indexCharArray      :: Ix ix => ByteArray ix -> ix -> Char
indexIntArray       :: Ix ix => ByteArray ix -> ix -> Int
indexWordArray      :: Ix ix => ByteArray ix -> ix -> Word
indexAddrArray      :: Ix ix => ByteArray ix -> ix -> Addr
indexFloatArray     :: Ix ix => ByteArray ix -> ix -> Float
indexDoubleArray    :: Ix ix => ByteArray ix -> ix -> Double
indexStablePtrArray :: Ix ix => ByteArray ix -> ix -> (StablePtr a)
```

```
sizeofByteArray     :: Ix ix => ByteArray ix -> Int
boundsOfByteArray   :: Ix ix => ByteArray ix -> (ix, ix)
```

Remarks:

- The operation `newByteArray` creates a byte array of length equal to the range of its indices *in bytes*.
- `sizeofByteArray` returns the size of the byte array, *in bytes*.
- Equality on byte arrays is value equality, not pointer equality (as is the case for its mutable variant.) Two byte arrays are equal if they're of the same length and they're pairwise equal.

4.4. `CCall`

This module is deprecated. Use the Foreign Function Interface instead.

The `CCall` module defines the classes `CCallable` and `CReturnable`, along with instances for the primitive types (`Int`, `Int#`, `Float`, `Float#` etc.) GHC knows to import this module if you use `_ccall_`, but if you need to define your own instances of these classes, you will need to import `CCall` explicitly.

4.5. `CTypes`

This module is part of the language-dependent part of the Foreign Function Interface (FFI) - for the language-independent part, see Section 4.9. It defines Haskell types that can hold the primitive types of C and can directly be used in foreign import and export declarations.

Every type has a constructor of the same name, which is currently exported, too. Given the large set of instances for each type, it is not clear if access to the concrete representation is really necessary. Some experience is needed before a final decision can be made in this respect.

4.5.1. Integral types

The following integral types have instances for the classes `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Typeable`, `Storable`, `Bounded`, `Real`, `Integral`, and `Bits`:

```
newtype CChar - char
newtype CSChar - signed char
newtype CUChar - unsigned char
newtype CShort - short
newtype CUShort - unsigned short
newtype CInt - int
newtype CUInt - unsigned int
newtype CLong - long
newtype CULong - unsigned long
newtype CLLong - long long
newtype CULLong - unsigned long long
```

4.5.2. Floating types

The following floating types have instances for the classes `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Typeable`, `Storable`, `Real`, `Fractional`, `Floating`, `RealFrac`, and `RealFloat`:

```
newtype CFloat - float
newtype CDouble - double
newtype CLDouble - long double
```

4.6. CTypesISO

This module is part of the language-dependent part of the Foreign Function Interface (FFI) - for the language-independent part, see Section 4.9. It defines Haskell types corresponding the most important ISO types of C that are not covered in the module `CTypes`.

Every type has a constructor of the same name, which is currently exported, too. Given the large set of instances for each type, it is not clear if access to the concrete representation is really necessary. Some experience is needed before a final decision can be made in this respect.

4.6.1. Integral types

The following integral types have instaces for the classes `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Typeable`, `Storable`, `Bounded`, `Real`, `Integral`, and `Bits`:

```
newtype CPtrdiff - ptrdiff_t
newtype CSize - size_t
newtype CWChar - wchar_t
newtype CSigAtomic - sig_atomic_t
```

4.6.2. Numeric types

The following numeric types have instances for the classes `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Typeable`, and `Storable`:

```
newtype CClock - clock_t
newtype CTime - time_t
```

4.6.3. Misc types

The following types have instances for the classes: ???

```
newtype CFile - FILE
newtype CFpos - fpos_t
newtype CJmpBuf - jmp_buf
```

4.7. Dynamic

The `Dynamic` library provides cheap-and-cheerful dynamic types for Haskell. A dynamically typed value is one which carries type information with it at run-time, and is represented here by the abstract type `Dynamic`. Values can be converted into `Dynamic` ones, which can then be combined and manipulated by the program using the operations provided over the abstract, dynamic type. One of these operations allows you to (try to) convert a dynamically-typed value back into a value with the same (monomorphic) type it had before converting it into a dynamically-typed value. If the dynamically-typed value isn't of the desired type, the coercion will fail.

The `Dynamic` library is capable of dealing with monomorphic types only; no support for polymorphic dynamic values, but hopefully that will be added at a later stage.

Examples where this library may come in handy (dynamic types, really - hopefully the library provided here will suffice) are: persistent programming, interpreters, distributed programming etc.

The following operations are provided over the `Dynamic` type:

```
data Dynamic - abstract, instance of: Show, Typeable
instance Show Dynamic

toDyn       :: Typeable a => a -> Dynamic
fromDyn     :: Typeable a => Dynamic -> a -> a
fromDynamic :: Typeable a => Dynamic -> Maybe a
```

- `toDyn` converts a value into a dynamic one, provided `toDyn` knows the (concrete) type representation of the value. The `Typeable` type class is used to encode this, overloading a function that returns the type representation of a value. More on this below.
- There's two ways of going from a dynamic value to one with a concrete type: `fromDyn`, tries to convert the dynamic value into a value with the same type as its second argument. If this fails, the default second argument is just returned. `fromDynamic` returns a `Maybe` type instead, `Nothing` coming back if the conversion was not possible.
- The `Dynamic` type has got a `Show` instance which returns a pretty printed string of the type of the dynamic value. (Useful when debugging).

4.7.1. Representing types

Haskell types are represented as terms using the `TypeRep` abstract type:

```
data TypeRep  - abstract, instance of: Eq, Show, Typeable
data TyCon    - abstract, instance of: Eq, Show, Typeable

mkTyCon  :: String  -> TyCon
mkAppTy  :: TyCon   -> [TypeRep] -> TypeRep
mkFunTy  :: TypeRep -> TypeRep   -> TypeRep
applyTy  :: TypeRep -> TypeRep   -> Maybe TypeRep
```

- `mkAppTy` applies a type constructor to a sequence of types, returning a type.
- `mkFunTy` is a special case of `mkAppTy`, applying the function type constructor to a pair of types.
- `applyTy` applies a type to a function type. If possible, the result type is returned.
- Type constructors are represented by the abstract type, `TyCon`.
- Most importantly, `TypeRep`s can be compared for equality. Type equality is used when converting a `Dynamic` value into a value of some specific type, comparing the type representation that the `Dynamic` value embeds with equality of the type representation of the type we're trying to convert the dynamically-typed value into.
- To allow comparisons between `TypeRep`s to be implemented efficiently, the *abstract* `TyCon` type is used, with the constructor function `mkTyCon` provided:

  ```
  mkTyCon :: String -> TyCon
  ```

An implementation of the `Dynamic` interface guarantees the following,

```
mkTyCon "a" == mkTyCon "a"
```

A really efficient implementation is possible if we guarantee/demand that the strings are unique, and for a particular type constructor, the application `mkTyCon` to the string that represents the type constructor is never duplicated. *Q: Would this constraint be unworkable in practice?*

- Both `TyCon` and `TypeRep` are instances of the `Show` type classes. To have tuple types be shown in infix form, the `Show` instance guarantees that type constructors consisting of `n`-commas, i.e., (`mkTyCon ",,,"`), is shown as an `(n+1)` tuple in infix form.

4.7.2. The Typeable class

To ease the construction of `Dynamic` values, we introduce the following type class to help working with `TypeRep`s:

```
class Typeable a where
  typeOf :: a -> TypeRep
```

- The `typeOf` function is overloaded to return the type representation associated with a type.
- *Important:* The argument to `typeOf` is only used to carry type information around so that overloading can be resolved. `Typeable` instances should never, ever look at this argument.
- The `Dynamic` library provides `Typeable` instances for all Prelude types and all types from the `lang` package (given that their component types are themselves Typeable). They are:

```
Prelude types:
   [a], (), (a,b), (a,b,c), (a,b,c,d), (a,b,c,d,e), (a->b),
   (Array a b), Bool, Char, (Complex a), Double, (Either a b),
   Float, Handle, Int, Integer, (IO a), (Maybe a), Ordering

Hugs/GHC types:
   Addr, AddrOff, Dynamic, ForeignObj, (IORef a),
   Int8, Int16, Int32, Int64, (ST s a), (StablePtr a),
   TyCon, TypeRep, Word8, Word16, Word32, Word64

GHC types:
   ArithException, AsyncException, (ByteArray i), CChar, CClock,
   CDouble, CFile, CFloat, CFpos, CInt, CJmpbuf, CLDouble,
   CLLong, CLong, CPtrdiff, CSChar, CShort, CSigAtomic, CSize,
   CTime, CUChar, CUInt, CULLong, CULong, CUShort, CWchar,
   Exception, (IOArray i e), (IOUArray i e), (MutableByteArray s i),
   PackedString, (STArray s i e), (STUArray s i e), (StableName a),
   (UArray i e), (Weak a)
```

Note: GHC's libraries currently contain the `Typeable` instances for the data types in the modules `Exception`, `CTypes`, and `CTypesISO` in those modules themselves. This is probably anyway the right way to go, `Dynamic` should only contain instances for Prelude types.

4.7.3. Utility functions

Operations for applying a dynamic function type to a dynamically typed argument are commonly useful, and also provided:

```
dynApply   :: Dynamic -> Dynamic -> Dynamic - unsafe.
dynApplyMb :: Dynamic -> Dynamic -> Maybe Dynamic
```

4.8. `Exception`

The Exception library provides an interface for raising and catching both built-in and user defined exceptions.

Exceptions are defined by the following (non-abstract) datatype:

```
- instance of Eq, Ord, Show, Typeable
data Exception
  = IOException  IOError - IO exceptions (from ’ioError’)
  | ArithException   ArithException - Arithmetic exceptions
  | ArrayException ArrayException  - Array-related exceptions
  | ErrorCall String - Calls to ’error’
  | NoMethodError       String - A non-existent method was invoked
  | PatternMatchFail String - A pattern match failed
  | RecSelError String - Selecting a non-existent field
  | RecConError String - Field missing in record construction
  | RecUpdError String - Record doesn’t contain updated field
  | AssertionFailed String - Assertions
  | DynException Dynamic - Dynamic exceptions
  | AsyncException AsyncException - Externally generated errors
  | PutFullMVar  - Put on a full MVar
  | BlockedOnDeadMVar - Blocking on a dead MVar
  | NonTermination

- instance of Eq, Ord, Show, Typeable
data ArithException
  = Overflow
  | Underflow
  | LossOfPrecision
  | DivideByZero
  | Denormal

- instance of Eq, Ord, Show, Typeable
data AsyncException
  = StackOverflow
  | HeapOverflow
  | ThreadKilled

- instance of Eq, Ord, Show, Typeable
data ArrayException
  = IndexOutOfBounds   String - out-of-range array access
  | UndefinedElement String - evaluating an undefined element
```

4.8.1. Kinds of exception

An implementation should raise the appropriate exception when once of the following conditions arises:

`IOException`

These are the standard IO exceptions from Haskell’s `IO` monad. IO Exceptions are raised by `IO.ioError`.

`ArithException`

Exceptions raised by arithmetic operations[1]:

`Overflow`

`Underflow`

`LossOfPrecision`

`DivisionByZero`

`Denormal`

`ArrayException`

Exceptions raised by array-related operations[2]:

`IndexOutOfBounds`

An attempt was made to index an array outside its declared bounds.

`UndefinedElement`

An attempt was made to evaluate an element of an array that had not been initialized.

`ErrorCall`

The `ErrorCall` exception is thrown by `error`. The `String` argument of `ErrorCall` is the string passed to `error` when it was called.

`NoMethodError`

An attempt was made to invoke a class method which has no definition in this instance, and there was no default definition given in the class declaration. GHC issues a warning when you compile an instance which has missing methods.

`PatternMatchFail`

A pattern matching failure. The `String` argument should contain a descriptive message including the function name, source file and line number.

`RecSelError`

A field selection was attempted on a constructor that doesn't have the requested field. This can happen with multi-constructor records when one or more fields are missing from some of the constructors. The `String` argument gives the location of the record selection in the source program.

`RecConError`

An attempt was made to evaluate a field of a record for which no value was given at construction time. The `String` argument gives the location of the record construction in the source program.

`RecUpdError`

An attempt was made to update a field in a record, where the record doesn't have the requested field. This can only occur with multi-constructor records, when one or more fields are missing from some of the constructors. The `String` argument gives the location of the record update in the source program.

`AssertionFailed`

This exception is thrown by the `assert` operation when the condition fails. The `String` argument contains the location of the assertion in the source program.

`DynException`

Dynamically typed exceptions, described in Section 4.8.5.

`AsyncException`

Asynchronous exceptions. These are described in more detail in Section 4.8.7. The types of asynchronous exception are:

`StackOverflow`

The current thread's stack exceeded its limit. Since an exception has been raised, the thread's stack will certainly be below its limit again, but the programmer should take remedial action immediately.

`HeapOverflow`

The program's heap is reaching its limit, and the program should take action to reduce the amount of live data it has[34].

`ThreadKilled`

This exception is raised by another thread calling `killThread` (see Section 2.2), or by the system if it needs to terminate the thread for some reason.

`PutFullMVar`

A call to `putMVar` (Section 2.4.3) was passed a full `MVar` .

`BlockedOnDeadMVar`

The current thread was executing a call to `takeMVar` (Section 2.4.3) that could never return, because there are no other references to this `MVar`.

`NonTermination`

The current thread is stuck in an infinite loop. This exception may or may not be thrown when the program is non-terminating.

4.8.2. Throwing exceptions

Exceptions may be thrown explicitly from anywhere:

```
throw :: Exception -> a
```

4.8.3. The `try` functions

There are several functions for catching and examining exceptions; all of them may only be used from within the `IO` monad. Firstly the `try` family of functions:

```
tryAll    :: a    -> IO (Either Exception a)
tryAllIO  :: IO a -> IO (Either Exception a)
try   :: (Exception -> Maybe b) -> a    -> IO (Either b a)
tryIO   :: (Exception -> Maybe b) -> IO a -> IO (Either b a)
```

The simplest version is `tryAll`. It takes a single argument, evaluates it (as if you'd applied `seq` to it), and returns either `Right a` if the evaluation succeeded with result `a`, or `Left e` if an exception was raised, where `e` is the exception. Note that due to Haskell's unspecified evaluation order, an expression may return one of several possible exceptions: consider the expression `error "urk" + 1 ‘div‘ 0`. Does `tryAll` return `Just (ErrorCall "urk")` or `Just (ArithError DivideByZero)`? The answer is "either": `tryAll` makes a non-deterministic choice about which exception to return. If you call it again, you might get a different exception back. This is ok, because `tryAll` is an IO computation.

`tryAllIO` is the same as `tryAll` except that the argument to evaluate is an `IO` computation. Don't try to use `tryAll` to catch exceptions in `IO` computations: in GHC an expression of type `IO a` is in fact a function, so evaluating it does nothing at all (and therefore raises no exceptions). Hence the need for `tryAllIO`, which runs `IO` computations properly.

The functions `try` and `tryIO` take an extra argument which is an *exception predicate*, a function which selects which type of exceptions we're interested in. The full set of exception predicates is given below:

```
justIoErrors :: Exception -> Maybe IOError
justArithExceptions  :: Exception -> Maybe ArithException
justErrors :: Exception -> Maybe String
```