Table of Contents

The Glasgow Haskell Compiler User's Guide, Version 5.02

The GHC Team

The Glasgow Haskell Compiler User's Guide, Version 5.02

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 Compiler (GHC): an interactive and batch compilation system for the Haskell 98 (http://www.haskell.org/) language.

GHC has two main components: an interactive Haskell interpreter (also known as GHCi), described in Chapter 3, and a batch compiler, described throughout Chapter 4. In fact, GHC consists of a single program which is just run with different options to provide either the interactive or the batch system.

The batch compiler can be used alongside GHCi: compiled modules can be loaded into an interactive session and used in the same way as interpreted code, and in fact when using GHCi most of the library code will be pre-compiled. This means you get the best of both worlds: fast pre-compiled library code, and fast compile turnaround for the parts of your program being actively developed.

GHC supports numerous language extensions, including concurrency, a foreign function interface, exceptions, type system extensions such as multi-parameter type classes, local universal and existential quantification, functional dependencies, scoped type variables and explicit unboxed types. These are all described in Chapter 7.

GHC has a comprehensive optimiser, so when you want to Really Go For It (and you've got time to spare) GHC can produce pretty fast code. Alternatively, the default option is to compile as fast as possible while not making too much effort to optimise the generated code (although GHC probably isn't what you'd describe as a fast compiler :-).

GHC's profiling system supports "cost centre stacks": a way of seeing the profile of a Haskell program in a call-graph like structure. See Chapter 5 for more details.

GHC comes with a large collection of libraries, with everything from parser combinators to networking. These are all described in *Haskell Libraries*.

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

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

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

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

glasgow-haskell-users:

This list is for GHC users to chat among themselves.

list email address:

<glasgow-haskell-users@haskell.org>

subscribe at:

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

admin email address:

<glasgow-haskell-users-admin@haskell.org>

list archives:

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

glasgow-haskell-bugs:

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

list email address:

<glasgow-haskell-bugs@haskell.org>

subscribe at:

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

admin email address:

<glasgow-haskell-bugs-admin@haskell.org>

list archives:

`http://www.haskell.org/pipermail/glasgow-haskell-bugs/`

cvs-ghc:

The hardcore GHC developers hang out here. This list also gets commit message from the CVS repository. There are several other similar lists for other parts of the CVS repository (eg. `cvs-hslibs`, `cvs-happy`, `cvs-hdirect` etc.)

list email address:

<cvs-ghc@haskell.org>

subscribe at:

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

admin email address:

<cvs-ghc-admin@haskell.org>

list archives:

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

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

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

1.2. Reporting bugs in GHC

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

To repot a bug, either:

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

1.2.1. What to put in a bug report

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

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

1.3. GHC version numbering policy

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

Stable Releases

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

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

Snapshots/unstable releases

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

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

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

The version number of your copy of GHC can be found by invoking `ghc` with the `--version` flag (see Section 4.4).

1.4. Release notes for version 5.02

1.4.1. User-visible compiler changes

- Majorly improved support for Windows platforms. Binary builds are now entirely freestanding. There is no longer any need to install Cygwin or Mingwin to use it. It's a one-click-install-and-off-you-go story now.
- Several small changes to bring GHC into line with the newest Haskell 98 report.
- RTS options can now be specified using the environment variable GHCRTS.

- The maximum heap size, set with `+RTS -Msize`, now defaults to unlimited. We recommend however that you set a maximum heap size appropriate for your machine using the GHCRTS variable.
- Thanks to the heroic efforts of Ken Shan <ken@digitas.harvard.edu>, GHC now works again on the Alpha architecture (just Tru64 at the moment), and many 64-bit bugs have been shaken out. Interactive mode and the native code generator do not currently work, sorry.
- New option: `-fno-code` which causes GHC to omit code generation and all future compilation stages (see Section 4.12.5).
- New option `-package-conf file`. This makes GHC read additional package descriptions from *file*. See Section 4.10.2.
- `+RTS -iseconds` flag reinstated for heap profiling. See Section 5.5.
- `-funbox-strict-fields` now works correctly. See Section 4.11.2.
- Many bug fixes related to implicit-parameter support.
- Data declarations with no constructors are allowed. This facilitates types whose only value is bottom, and which have no representation.
- You can disconnect numeric syntax from the Prelude syntax. Doing this means you can define your own arithmetic. The relevant flag is `-fno-implicit-prelude`. See Section 7.1.
- Versions 5.02.1 and later: `-O2-for-C` has been removed. It sometimes caused gcc to generate incorrect code, and rarely gives any benefit. If you absolutely insist on potentially shooting yourself in the foot, you can get the same effect by using the flag `-optc-O2`.
- You can disconnect numeric syntax from the Prelude syntax. Doing this means you can define your own arithmetic. The relevant flag is `-fno-implicit-prelude`. See Section 7.1.

1.4.2. User-visible interpreter (GHCi) changes

- GHCi now works on Windows.
- Partial FFI support in GHCi. At the moment, foreign import (static and dynamic) are supported on x86 and sparc platforms.
- New interpreter command `:add`, to add modules to the current bunch. See Section 3.6.
- New interpreter command `:info`, to provide information on values, types and classes, a la Hugs. See Section 3.6.
- Multiple target modules are supported. You may specify multiple modules for the `:add` and `:load` commands. From the command line, you may specify multiple modules when starting GHCi and when compiling in `--make` mode. See Section 3.6.
- New flags `-ignore-dot-ghci` and `-read-dot-ghci`. See Section 3.8.

- When starting GHCi from the command line, you may now use the flags `-Llibrary-path` and `-llibrary-name` to specify libraries to be loaded into the interactive session. See Section 3.5.

1.4.3. User-visible library changes

- The `SocketPrim` and `BSD` modules in the `net` package have been re-written to use the FFI, be more portable, and be more thread-safe. There are minor user-visible changes: the functions `writeSocket`, `readSocket` and `readSocketAll` have been removed (use `socketToHandle` and `Handle` operations instead).
- New function: `Exception.throwDynTo`. See Section 5.12.5 in *Haskell Libraries*.
- `BlockedOnDeadMVar` and `ThreadKilled` exceptions no longer generate any output by default for forked threads.
- `Weak.addForeignFinalizer` is deprecated; use the `ForeignPtr` library instead.
- The I/O library has been completely rewritten, using the new FFI libraries and **hsc2hs**. The main improvement is to the way streams are handled: simultaneous read and write with multiple threads using a socket or FIFO is now possible, whereas before it was necessary to use two separate handles.
- As a result of the I/O rewrite, the extensions `IOExts.hConnectTo` and `IOExts.withHandleFor` have been removed. Also, `hGetBuf` and `hPutBuf` have been removed, and `hGetBufFull` and `hPutBufFull` have been renamed to `hGetBuf` and `hPutBuf` respectively (similary for the `BA` versions of these functions).
- `System.exitWith` now throws `ExitException` rather than causing the program to exit immediately. `ExitException` is by default caught by the top level exception handler in the main thread, where it causes the program to exit. Hence, calling `System.exitWith` in GHCi no longer causes GHCi itself to exit.
- New function: `MVar.addMVarFinalizer` (see Section 3.4.3 in *Haskell Libraries*).
- New module `SystemExts` for useful system-ish functionality (see Section 5.37 in *Haskell Libraries*).
- `SocketPrim`: added instances of `Eq` and `Show` for the `Socket` type.
- The implementation of `MarsalAlloc.allocaBytes` now uses GHC's internal allocator and is much faster than before. As a result, the `UnsafeCString` type and functions are no longer necessary and have been removed from `CString`.

1.4.4. New experimental features

- Partial support for hierarchical module names.

- Mucho hacking on the .NET code generator, including some FFI extensions for .NET interop. It's still severely b0rk3d, so won't do anything useful. Yet.
- Emission of external Core format. The goal is for other tools to be able to grab the Core resulting from GHC's front end manglings and optimisations. Core format is formally defined by the document *`An External Representation of the GHC Core Language`* in the GHC repository. We also have sample tools for reading, writing and typechecking external Core. Ultimately we plan that Core files can also be read by GHC. The relevant flag is `-fext-core`.
- A compacting garbage collector has been added. It isn't on by default, because it is somewhat slower than the existing copying collector, but can be automatically enabled when memory gets tight by setting the maximum heap size (see Section 4.16.2). Compaction works together with the existing generational scheme: it is only used on the oldest generation. Despite the "experimental" classification, we've jumped up and down quite significantly on this code, and it seems stable.

1.4.5. Internal changes

- The implementation of the `Array`, `IArray` and `MArray` libraries have been rewritten so that bulk operations like `listArray` and `elems` do less range checking and hence go faster.
- The `CPUTime` library has been replaced by one written using the FFI. No externally visible changes.
- The compiler has a new strictness-and-absence analyser, which is alleged to help it generate better code.

Chapter 2. Installing GHC

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

This guide is in several parts:

- Installing on Unix-a-likes (Section 2.1).
- Installing on Windows (Section 2.2).
- The layout of installed files (Section 2.3). You don't need to know this to install GHC, but it's useful if you are changing the implementation.
- Installing or building the documentation (Section 2.4).

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 follow.

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

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

   `platform`

   the platform you are going to install for.

`bindir`

the directory in which to install user-invokable binaries.

`libdir`

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

`datadir`

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

`infodir`

the directory in which to install Emacs info files.

`htmldir`

the directory in which to install HTML documentation.

`dvidir`

the directory in which to install DVI documentation.

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

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

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

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

2.1.2.2. What bundles there are

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

`prof`:

Profiling with cost-centres. You probably want this.

`par`:

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

`gran`:

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

`ticky`:

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

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

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

2.1.2.3. Testing that GHC seems to be working

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

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

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

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

Now run it:

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

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

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

2.2. Installing on Windows

Getting the Glasgow Haskell Compiler (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 use GHC is included in the InstallShield.

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

2.2.1. Installing GHC

Download the latest GHC distribution (ghc-5.02 InstallShield installer, 27M) from haskell.org (http://www.haskell.org/ghc/download_ghc_502.html).

When the installer has completed, make sure you add the location of the ghc `bin/` directory to your path, as directed in the final dialog of the installer. You need to do this in order to bring the various GHC binaries into scope.

You can freely move the GHC tree once you've installed it just by copying the `ghc-x.yy` directory. You might want to do this in order to use GHC with tools that don't like spaces in paths (GHC is installed in `\Program Files\Glasgow Haskell Compiler` by default.

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 9).

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 the **rm** or **mv**.

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

2.3. The layout of installed files

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

GHC is installed in two directory trees:

Binary directory

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

Library directory,

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

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

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

2.3.1. Layout of the library directory

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

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

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

    unlit                  Remove literate markup

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

    ghc-x.xx               GHC executable [Unix only]

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

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

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

        crt0.o              Standard
   ..etc..     binaries

        libmingw32.a        Standard
   ..etc..     libraries

        *.h                 Include files

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

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

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

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

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

Note that:

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

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

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

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

2.4. 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.

2.4.1. 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`.

2.4.2. 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.

2.4.3. 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`).

2.4.4. Installing the DocBook tools from source

2.4.4.1. 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`.

2.4.4.2. 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`.

2.4.5. 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.

2.4.6. 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 GHCi

GHCi[1] is GHC's interactive environment, in which Haskell expressions can be interactively evaluated and programs can be interpreted. If you're famililar with Hugs, then you'll be right at home with GHCi. However, GHCi also has support for interactively loading compiled code, as well as supporting all[2]the language extensions that GHC provides.

3.1. Introduction to GHCi

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

```
$ ghci
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 5.00, For Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package std ... linking ... done.
Prelude>
```

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

```
 Commands available from the prompt:
   <stmt>              evaluate/run <stmt>
   :cd <dir>           change directory to <dir>
   :def <cmd> <expr>   define a macro :<cmd>
   :help, :?           display this list of commands
   :load <filename>    load a module (and it dependents)
   :module <mod>       set the context for expression evaluation to <mod>
   :reload             reload the current module set
   :set <option> ...   set options
   :type <expr>        show the type of <expr>
   :unset <option> ... unset options
   :quit               exit GHCi
   :!<command>         run the shell command <command>
 Options for ':set' and ':unset':
    +r                 revert top-level expressions after each evaluation
    +s                 print timing/memory stats after each evaluation
    +t                 print type after evaluation
    -<flag>            most GHC command line flags can also be set here
                         (eg. -v2, -fglasgow-exts, etc.)
```

We'll explain most of these commands as we go along. For Hugs users: many things work the same as in Hugs, so you should be able to get going straight away.

Haskell expressions can be typed at the prompt:

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

GHCi interprets the whole line as an expression to evaluate. The expression may not span several lines - as soon as you press enter, GHCi will attempt to evaluate it.

3.2. Loading source files

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

```
main = print (fac 20)

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

To load a Haskell source file into GHCi, use the `:load` command:

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

GHCi has loaded the `Main` module, and the prompt has changed to "`Main>`" to indicate that the current context for expressions typed at the prompt is the `Main` module we just loaded. So we can now type expressions involving the functions from `Main.hs`:

```
Main> fac 17
355687428096000
```

Loading a multi-module program is just as straightforward; just give the name of the "topmost" module to the `:load` command (hint: `:load` can be abbreviated to `:l`). The topmost module will normally be `Main`, but it doesn't have to be. GHCi will discover which modules are required, directly or indirectly, by the topmost module, and load them all in dependency order.

3.2.1. Modules vs. filenames

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

There is one exception to this general rule: when you load a program with `:load`, or specify it when you invoke `ghci`, you can give a filename rather than a module name. This filename is loaded if it exists, and it may contain any module you like. This is particularly convenient if you have several `Main` modules in the same directory and you can't call them all `Main.hs`.

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

ghci -i*dir*$_1$:...:*dir*$_n$

or it can be set using the `:set` command from within GHCi (see Section 3.7.2)[3]

One consequence of the way that GHCi follows dependencies to find modules to load is that every module must have a source file. The only exception to the rule is modules that come from a package, including the `Prelude` and standard libraries such as `IO` and `Complex`. If you attempt to load a module for which GHCi can't find a source file, even if there are object and interface files for the module, you'll get an error message.

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

3.2.2. Making changes and recompilation

If you make some changes to the source code and want GHCi to recompile the program, give the `:reload` command. The program will be recompiled as necessary, with GHCi doing its best to avoid actually recompiling modules if their external dependencies haven't changed. This is the same mechanism we use to avoid re-compiling modules in the batch compilation setting (see Section 4.9.4).

3.3. Loading compiled code

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

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

When loading up source files with `:load`, GHCi looks for any corresponding compiled object files, and will use one in preference to interpreting the source if possible. For example, suppose we have a 4-module program consisting of modules A, B, C, and D. Modules B and C both import D only, and A imports both B & C:

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

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

```
Prelude> :! ghc -c D.hs
Prelude> :load A
Skipping  D                ( D.hs, D.o )
Compiling C                ( C.hs, interpreted )
Compiling B                ( B.hs, interpreted )
Compiling A                ( A.hs, interpreted )
Ok, modules loaded: A, B, C, D.
Main>
```

In the messages from the compiler, we see that it skipped D, and used the object file `D.o`. The message `Skipping` *`module`* indicates that compilation for *`module`* isn't necessary, because the source and everything it depends on is unchanged since the last compilation.

If we now modify the source of D (or pretend to: using Unix command `touch` on the source file is handy for this), the compiler will no longer be able to use the object file, because it might be out of date:

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

Note that module D was compiled, but in this instance because its source hadn't really changed, its interface remained the same, and the recompilation checker determined that A, B and C didn't need to be recompiled.

So let's try compiling one of the other modules:

```
Main> :! ghc -c C.hs
Main> :load A
Compiling D                ( D.hs, interpreted )
Compiling C                ( C.hs, interpreted )
Compiling B                ( B.hs, interpreted )
Compiling A                ( A.hs, interpreted )
Ok, modules loaded: A, B, C, D.
```

We didn't get the compiled version of C! What happened? Well, in GHCi a compiled module may only depend on other compiled modules, and in this case C depends on D, which doesn't have an object file, so GHCi also rejected C's object file. Ok, so let's also compile D:

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

Nothing happened! Here's another lesson: newly compiled modules aren't picked up by `:reload`, only `:load`:

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

HINT: since GHCi will only use a compiled object file if it can sure that the compiled version is up-to-date, a good technique when working on a large program is to occasionally run `ghc --make` to compile the whole project (say before you go for lunch :-), then continue working in the interpreter. As you modify code, the new modules will be interpreted, but the rest of the project will remain compiled.

3.4. Interactive evaluation at the prompt

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

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

What actually happens is that GHCi typechecks the expression, and if it doesn't have an `IO` type, then it transforms it as follows: an expression *e* turns into

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

which is then run as an IO-action.

Hence, the original expression must have a type which is an instance of the `Show` class, or GHCi will complain:

```
Prelude> id
```

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

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

3.4.1. What's really in scope at the prompt?

When you type an expression at the prompt, what identifiers and types are in scope? GHCi has a concept of a *context* module, which can be set using the `:module` command.

The context module is shown in the prompt: for example, the prompt `Prelude>` indicates that the current context for evaluating expressions is the Haskell `Prelude` module. The Prelude is the default context when you start up GHCi.

Exactly which entities are in scope in a given context depends on whether the context module is compiled or interpreted:

- If the context module is interpreted, then everything that was in scope during compilation of that module is also in scope at the prompt, i.e. all the imports and any top-level functions, types and classes defined in that module.
- If the context module comes from a package, or is otherwise compiled, then only the exports of that module are in scope at the prompt. So for example, when the current context module is `Prelude`, everything the `Prelude` exports is in scope, but if we switch context to eg. `Time`, then everything from the `Prelude` is now invisible.

The reason for this unfortunate distinction is boring: for a compiled module when the source isn't available, the compiler has no way of knowing what was in scope when the module was compiled (and we don't store this information in the interface file). However, in practice it shouldn't be a problem: if you want both `Time` and `Prelude` in scope at the same time, just create a file containing the line `import Time` and load it into GHCi.

To make life slightly easier, the GHCi prompt also behaves as if there is an implicit `import qualified` declaration for every module in every package, and every module currently loaded into GHCi. So in the above example where the `Prelude` was invisible, we can always get at `Prelude` identifiers by qualifying them, eg. `Prelude.map`.

3.4.2. Using `do`-notation at the prompt

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

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

Here's an example:

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

The statement `x <- return 42` means "execute `return 42` in the `IO` monad, and bind the result to `x`". We can then use `x` in future statements, for example to print it as we did above.

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

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

An important difference between the two types of binding is that the monadic bind (`p <- e`) is *strict* (it evaluates `e`), whereas with the `let` form, the expression isn't evaluated immediately:

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

Any exceptions raised during the evaluation or execution of the statement are caught and printed by the GHCi command line interface (see Section 5.12 in *Haskell Libraries* for more information on GHC's Exception support).

Every new binding shadows any existing bindings of the same name, including entities that are in scope in the current module context.

WARNING: temporary bindings introduced at the prompt only last until the next `:load` or `:reload` command, at which time they will be simply lost. However, they do survive a change of context with `:module`: the temporary bindings just move to the new location.

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

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

3.4.3. The `it` variable

Whenever an expression (or a non-binding statement, to be precise) is typed at the prompt, GHCi implicitly binds its value to the variable `it`. For example:

```
Prelude> 1+2
3
Prelude> it * 2
6
```

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

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

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

If the expression was of type `IO a` for some `a`, then `it` will be bound to the result of the `IO` computation, which is of type `a`. eg.:

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

The corresponding translation for an IO-typed *e* is

```
it <- e
```

Note that `it` is shadowed by the new value each time you evaluate a new expression, and the old value of `it` is lost.

3.5. Invoking GHCi

GHCi is invoked with the command `ghci` or `ghc --interactive`. One or more modules or filenames can also be specified on the command line; this instructs GHCi to load the specified modules or filenames (and all the modules they depend on), just as if you had said `:load` *modules* at the GHCi prompt (see Section 3.6). For example, to start GHCi and load the program whose topmost module is in the file `Main.hs`, we could say:

```
$ ghci Main.hs
```

Most of the command-line options accepted by GHC (see Chapter 4) also make sense in interactive mode. The ones that don't make sense are mostly obvious; for example, GHCi doesn't generate interface files, so options related to interface file generation won't have any effect.

3.5.1. Packages

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

```
$ ghci -package text
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 5.00, For Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package std ... linking ... done.
Loading package lang ... linking ... done.
Loading package text ... linking ... done.
Prelude>
```

Note that GHCi also loaded the `lang` package even though we didn't ask for it: that's because the `text` package makes use of one or more of the modules in `lang`, and therefore has a dependency on it.

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

```
Prelude> :set -package name
```

But note that doing this will cause all currently loaded modules to be unloaded, and you'll be dumped back into the Prelude.

3.5.2. Extra libraries

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

```
$ ghci -lm
```

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

- Paths specified using the `-Lpath` command-line option,
- the standard library search path for your system, which on some systems may be overriden by setting the `LD_LIBRARY_PATH` environment variable.

On systems with `.dll`-style shared libraries, the actual library loaded will be `lib.dll`. Again, GHCi will signal an error if it can't find the library.

GHCi can also load plain object files (`.o` or `.obj` depending on your platform) from the command-line. Just add the name the object file to the command line.

3.6. GHCi commands

GHCi commands all begin with ‘:’ and consist of a single command name followed by zero or more parameters. The command name may be abbreviated, as long as the abbreviation is not ambiguous. All of the builtin commands, with the exception of `:unset` and `:undef`, may be abbreviated to a single letter.

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

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

`:cd` *`dir`*

Changes the current working directory to *`dir`*. A ‘<732>’ symbol at the beginning of *`dir`* will be replaced by the contents of the environment variable `HOME`.

`:def` *`name expr`*

The command `:def` *`name expr`* defines a new GHCi command `:`*`name`*, implemented by the Haskell expression *`expr`*, which must have type `String -> IO String`. When `:`*`name args`* is typed at the prompt, GHCi will run the expression `(`*`name args`*`)`, take the resulting `String`, and feed it back into GHCi as a new sequence of commands. Separate commands in the result must be separated by ‘`\n`’.

That’s all a little confusing, so here’s a few examples. To start with, here’s a new GHCi command which doesn’t take any arguments or produce any results, it just outputs the current date & time:

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

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

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

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

```
Prelude> :def make (\_ -> return ":! ghc --make Main")
```

`:help`
`:?`

Displays a list of the available commands.

`:info` *`name`* ...

Displays information about the given name(s). For example, if *`name`* is a class, then the class methods and their types will be printed; if *`name`* is a type constructor, then its definition will be printed; if *`name`* is a function, then its type will be printed. If *`name`* has been loaded from a source file, then GHCi will also display the location of its definition in the source.

`:load` *`module`* ...

Recursively loads the specified *`module`*s, and all the modules they depend on. Here, each *`module`* must be a module name or filename, but may not be the name of a module in a package.

All previously loaded modules, except package modules, are forgotten. The new set of modules is known as the *target set*.

After a `:load` command, the current context is set to:

- *`module`*, if it was loaded successfully, or
- the most recently successfully loaded module, if any other modules were loaded as a result of the current `:load`, or
- `Prelude` otherwise.

`:module` *`module`*

Sets the current context for statements typed at the prompt to *`module`*, which must be a module name which is already loaded or in a package. See Section 3.4.1 for more information on what effect the context has on what entities are in scope at the prompt.

`:quit`

Quits GHCi. You can also quit by typing a control-D at the prompt.

`:reload`

Attempts to reload the current target set (see `:load`) if any of the modules in the set, or any dependent module, has changed. Note that this may entail loading new modules, or dropping modules which are no longer indirectly required by the target.

`:set` [*`option`*...]

Sets various options. See Section 3.7 for a list of available options. The `:set` command by itself shows which options are currently set.

`:type` *`expression`*

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

`:undef` *`name`*

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

`:unset` *`option`*...

Unsets certain options. See Section 3.7 for a list of available options.

`:!` *command*...

Executes the shell command *command*.

3.7. The `:set` command

The `:set` command sets two types of options: GHCi options, which begin with ‘`+`” and “command-line” options, which begin with ‘-’.

3.7.1. GHCi options

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

The available GHCi options are:

`+r`

Normally, any evaluation of top-level expressions (otherwise known as CAFs or Constant Applicative Forms) in loaded modules is retained between evaluations. Turning on `+r` causes all evaluation of top-level expressions to be discarded after each evaluation (they are still retained *during* a single evaluation).

This option may help if the evaluated top-level expressions are consuming large amounts of space, or if you need repeatable performance measurements.

`+s`

Display some stats after evaluating each expression, including the elapsed time and number of bytes allocated. NOTE: the allocation figure is only accurate to the size of the storage manager’s allocation area, because it is calculated at every GC. Hence, you might see values of zero if no GC has occurred.

`+t`

Display the type of each variable bound after a statement is entered at the prompt. If the statement is a single expression, then the only variable binding will be for the variable ‘`it`’.

3.7.2. Setting GHC command-line options in GHCi

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

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

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

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

Section 4.19 lists the reverse for each option where applicable.

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

3.8. The `.ghci` file

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

The `.ghci` in your home directory is most useful for turning on favourite options (eg. `:set +s`), and defining useful macros. Placing a `.ghci` file in a directory with a Haskell project is a useful way to set certain project-wide options so you don't have to type them everytime you start GHCi: eg. if your project uses GHC extensions and CPP, and has source files in three subdirectories A B and C, you might put the following lines in `.ghci`:

```
:set -fglasgow-exts -cpp
:set -iA:B:C
```

(Note that strictly speaking the `-i` flag is a static one, but in fact it works to set it using `:set` like this. The changes won't take effect until the next `:load`, though.)

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

`-ignore-dot-ghci`

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

`-read-dot-ghci`

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

3.9. FAQ and Things To Watch Out For

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

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

`System.getArgs` returns GHCi's command line arguments!

Yes, it does.

The interpreter can't load modules with foreign export declarations!

Unfortunately not. We haven't implemented it yet. Please compile any offending modules by hand before loading them into GHCi.

`-O` doesn't work with GHCi!

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

Unboxed tuples don't work with GHCi

That's right. You can always compile a module that uses unboxed tuples and load it into GHCi, however. (Incidentally the previous point, namely that `-O` is incompatible with GHCi, is because the bytecode compiler can't deal with unboxed tuples).

Concurrent threads don't carry on running when GHCi is waiting for input.

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

After using `getContents`, I can't use `stdin` again until I do `:load` or `:reload`.

This is the defined behaviour of `getContents`: it puts the stdin Handle in a state known as *semi-closed*, wherein any further I/O operations on it are forbidden. Because I/O state is retained between computations, the semi-closed state persists until the next `:load` or `:reload` command.

You can make `stdin` reset itself after every evaluation by giving GHCi the command `:set +r`. This works because `stdin` is just a top-level expression that can be reverted to its unevaluated state in the same way as any other top-level expression (CAF).

Notes

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

3. Note that in GHCi, and `--make` mode, the `-i` option is used to specify the search path for *source* files, whereas in standard batch-compilation mode the `-i` option is used to specify the search path for interface files, see Section 4.9.2.

Chapter 4. Using GHC

GHC can work in one of three “modes”:

ghc —interactive

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

ghc —make

In this mode, GHC will build a multi-module Haskell program automatically, figuring out dependencies for itself. If you have a straightforward Haskell program, this is likely to be much easier, and faster, than using **make**.

ghc [-E | -C | -S | -c]

This is the traditional batch-compiler mode, in which GHC can compile source files one at a time, or link objects together into an executable.

4.1. Options overview

GHC’s behaviour is controlled by *options*, which for historical reasons are also sometimes referred to as command-line flags or arguments. Options can be specified in three ways:

4.1.1. Command-line arguments

An invocation of GHC takes the following form:

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

Command-line arguments are either options or file names.

Command-line options begin with `-`. They may *not* be grouped: `-vO` is different from `-v -O`. Options need not precede filenames: e.g., `ghc *.o -o foo`. All options are processed and then applied to all files; you cannot, for example, invoke `ghc -c -O1 Foo.hs -O2 Bar.hs` to apply different optimisation levels to the files `Foo.hs` and `Bar.hs`.

4.1.2. Command line options in source files

Sometimes it is useful to make the connection between a source file and the command-line options it requires quite tight. For instance, if a Haskell source file uses GHC extensions, it will always need to be compiled with the `-fglasgow-exts` option. Rather than maintaining the list of per-file options in a `Makefile`, it is possible to do this directly in the source file using the `OPTIONS` pragma :

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

`OPTIONS` pragmas are only looked for at the top of your source files, upto the first (non-literate,non-empty) line not containing `OPTIONS`. Multiple `OPTIONS` pragmas are recognised. Note that your command shell does not get to the source file options, they are just included literally in the array of command-line arguments the compiler driver maintains internally, so you'll be desperately disappointed if you try to glob etc. inside `OPTIONS`.

NOTE: the contents of OPTIONS are prepended to the command-line options, so you *do* have the ability to override OPTIONS settings via the command line.

It is not recommended to move all the contents of your Makefiles into your source files, but in some circumstances, the `OPTIONS` pragma is the Right Thing. (If you use `-keep-hc-file-too` and have OPTION flags in your module, the OPTIONS will get put into the generated .hc file).

4.1.3. Setting options in GHCi

Options may also be modified from within GHCi, using the `:set` command. See Section 3.7 for more details.

4.2. Static vs. Dynamic options

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

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

4.3. Meaningful file suffixes

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

`.lhs`

A "literate Haskell" module.

`.hs`

A not-so-literate Haskell module.

`.hi`

A Haskell interface file, probably compiler-generated.

`.hc`

Intermediate C file produced by the Haskell compiler.

`.c`

A C file not produced by the Haskell compiler.

`.s`

An assembly-language source file, usually produced by the compiler.

`.o`

An object file, produced by an assembler.

Files with other suffixes (or without suffixes) are passed straight to the linker.

4.4. Help and verbosity options

`--help`
`-?`

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

`-v`

The `-v` option makes GHC *verbose*: it reports its version number and shows (on stderr) exactly how it invokes each phase of the compilation system. Moreover, it passes the `-v` flag to most phases; each reports its version number (and possibly some other information).

Please, oh please, use the `-v` option when reporting bugs! Knowing that you ran the right bits in the right order is always the first thing we want to verify.

`-v`*n*

To provide more control over the compiler's verbosity, the `-v` flag takes an optional numeric argument. Specifying `-v` on its own is equivalent to `-v3`, and the other levels have the following meanings:

`-v0`

Disable all non-essential messages (this is the default).

`-v1`

Minimal verbosity: print one line per compilation (this is the default when `--make` or `--interactive` is on).

`-v2`

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

`-v3`

The same as `-v2`, except that in addition the full command line (if appropriate) for each compilation phase is also printed.

`-v4`

The same as `-v3` except that the intermediate program representation after each compilation phase is also printed (excluding preprocessed and C/assembly files).

`--version`

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

`--numeric-version`

Print GHC's numeric version number only.

`--print-libdir`

Print the path to GHC's library directory. This is the top of the directory tree containing GHC's libraries, interfaces, and include files (usually something like `/usr/local/lib/ghc-5.02` on Unix). This is the value of `$libdir`in the package configuration file (see Section 4.10).

4.5. Using ghc `--make`

When given the `--make` option, GHC will build a multi-module Haskell program by following dependencies from a single root module (usually `Main`). For example, if your `Main` module is in a file called `Main.hs`, you could compile and link the program like this:

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

The command line must contain one source file or module name; GHC will figure out all the modules in the program by following the imports from this initial module. It will then attempt to compile each module which is out of date, and finally if the top module is `Main`, the program will also be linked into an executable.

The main advantages to using `ghc --make` over traditional `Makefile`s are:

- GHC doesn't have to be restarted for each compilation, which means it can cache information between compilations. Compiling a muli-module program with `ghc --make` can be up to twice as fast as running `ghc` individually on each source file.
- You don't have to write a `Makefile`.
- GHC re-calculates the dependencies each time it is invoked, so the dependencies never get out of sync with the source.

Any of the command-line options described in the rest of this chapter can be used with `--make`, but note that any options you give on the command line will apply to all the source files compiled, so if you want any options to apply to a single source file only, you'll need to use an `OPTIONS` pragma (see Section 4.1.2).

If the program needs to be linked with additional objects (say, some auxilliary C code), these can be specified on the command line as usual.

Note that GHC can only follow dependencies if it has the source file available, so if your program includes a module for which there is no source file, even if you have an object and an interface file for the module, then GHC will complain. The exception to this rule is for package modules, which may or may not have source files.

4.6. GHC without `--make`

Without `--make`, GHC will compile one or more source files given on the command line.

The first phase to run is determined by each input-file suffix, and the last phase is determined by a flag. If no relevant flag is present, then go all the way through linking. This table summarises:

Phase of the compilation system	Suffix saying "start here"	Flag saying "stop after"	(suffix of) output file
literate pre-processor	`.lhs`	-	`.hs`
C pre-processor (opt.)	`.hs (with -cpp)`	`-E`	`.hspp`
Haskell compiler	`.hs`	`-C, -S`	`.hc, .s`
C compiler (opt.)	`.hc or .c`	`-S`	`.s`
assembler	`.s`	`-c`	`.o`

Phase of the compilation system	Suffix saying “start here”	Flag saying “stop after”	(suffix of) output file
linker	*other*	-	a.out

Thus, a common invocation would be: `ghc -c Foo.hs`

Note: What the Haskell compiler proper produces depends on whether a native-code generator is used (producing assembly language) or not (producing C). See Section 4.12.5 for more details.

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

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

4.7. Re-directing the compilation output(s)

`-o`

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

Note: this “feature” can be counterintuitive: **ghc -C -o foo.o foo.hs** will put the intermediate C code in the file `foo.o`, name notwithstanding!

`-odir`

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

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

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

Note that the `-odir` option does *not* affect where the interface files are put. In the above example, they would still be put in `parse/Foo.hi`, `parse/Bar.hi`, and `gurgle/Bumble.hi`.

`-ohi` *`file`*

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

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

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

`-hidir` *`directory`*

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

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

EXOTICA: The `-osuf` *`suffix`* will change the `.o` file suffix for object files to whatever you specify. We use this when compiling libraries, so that objects for the profiling versions of the libraries don't clobber the normal ones.

Similarly, the `-hisuf` *`suffix`* will change the `.hi` file suffix for non-system interface files (see Section 4.9.3).

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

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...

4.7.1. Keeping Intermediate Files

The following options are useful for keeping certain intermediate files around, when normally GHC would throw these away after compilation:

`-keep-hc-files`

Keep intermediate `.hc` files when doing `.hs`-to-`.o` compilations via C (NOTE: `.hc` files aren't generated when using the native code generator, you may need to use `-fvia-C` to force them to be produced).

`-keep-s-files`

Keep intermediate `.s` files.

`-keep-raw-s-files`

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

`-keep-tmp-files`

Instructs the GHC driver not to delete any of its temporary files, which it normally keeps in `/tmp` (or possibly elsewhere; see Section 4.7.2). Running GHC with `-v` will show you what temporary files were generated along the way.

4.7.2. Redirecting temporary files

`-tmpdir`

If you have trouble because of running out of space in `/tmp` (or wherever your installation thinks temporary files should go), you may use the `-tmpdir <dir>` option to specify an alternate directory. For example, `-tmpdir .` says to put temporary files in the current working directory.

Alternatively, use your `TMPDIR` environment variable. Set it to the name of the directory where temporary files should be put. GCC and other programs will honour the `TMPDIR` variable as well.

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

4.8. Warnings and sanity-checking

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

`-W`:

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

`-w`:

Turns off all warnings, including the standard ones.

`-Wall`:

Turns on all warning options.

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

`-fwarn-deprecations`:

Causes a warning to be emitted when a deprecated function or type is used. Entities can be marked as deprecated using a pragma, see Section 7.15.7.

`-fwarn-duplicate-exports`:

Have the compiler warn about duplicate entries in export lists. This is useful information if you maintain large export lists, and want to avoid the continued export of a definition after you've deleted (one) mention of it in the export list.

This option is on by default.

`-fwarn-hi-shadowing`:

Causes the compiler to emit a warning when a module or interface file in the current directory is shadowing one with the same module name in a library or other directory.

`-fwarn-incomplete-patterns`:

Similarly for incomplete patterns, the function `g` below will fail when applied to non-empty lists, so the compiler will emit a warning about this when `-fwarn-incomplete-patterns` is enabled.

```
g [] = 2
```

This option isn't enabled be default because it can be a bit noisy, and it doesn't always indicate a bug in the program. However, it's generally considered good practice to cover all the cases in your functions.

`-fwarn-misc`:

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

`-fwarn-missing-fields`:

This option is on by default, and warns you whenever the construction of a labelled field constructor isn't complete, missing initializers for one or more fields. While not an error (the missing fields are initialised with bottoms), it is often an indication of a programmer error.

`-fwarn-missing-methods`:

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

`-fwarn-missing-signatures`:

If you would like GHC to check that every top-level function/value has a type signature, use the `-fwarn-missing-signatures` option. This option is off by default.

`-fwarn-name-shadowing`:

This option causes a warning to be emitted whenever an inner-scope value has the same name as an outer-scope value, i.e. the inner value shadows the outer one. This can catch typographical errors that turn into hard-to-find bugs, e.g., in the inadvertent cyclic definition `let x = ... x ... in`.

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

`-fwarn-overlapping-patterns`:

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

```
f :: String -> Int
f []     = 0
f (_:xs) = 1
f "2"    = 2
```

where the last pattern match in `f` won't ever be reached, as the second pattern overlaps it. More often than not, redundant patterns is a programmer mistake/error, so this option is enabled by default.

`-fwarn-simple-patterns`:

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

`-fwarn-type-defaults`:

Have the compiler warn/inform you where in your source the Haskell defaulting mechanism for numeric types kicks in. This is useful information when converting code from a context that assumed one default into one with another, e.g., the 'default default' for Haskell 1.4 caused the otherwise unconstrained value `1` to be given the type `Int`, whereas Haskell 98 defaults it to `Integer`. This may lead to differences in performance and behaviour, hence the usefulness of being non-silent about this.

This warning is off by default.

`-fwarn-unused-binds`:

Report any function definitions (and local bindings) which are unused. For top-level functions, the warning is only given if the binding is not exported.

`-fwarn-unused-imports`:

Report any objects that are explicitly imported but never used.

`-fwarn-unused-matches`:

Report all unused variables which arise from pattern matches, including patterns consisting of a single variable. For instance `f x y = []` would report `x` and `y` as unused. To eliminate the warning, all unused variables can be replaced with wildcards.

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

4.9. Separate compilation

This section describes how GHC supports separate compilation.

4.9.1. Interface files

When GHC compiles a source file `A.hs` which contains a module `A`, say, it generates an object `A.o`, *and* a companion *interface file* `A.hi`. The interface file is 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: In general, the name of a file containing module `M` should be named `M.hs` or `M.lhs`. The only exception to this rule is module `Main`, which can be placed in any file.

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

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

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

4.9.2. Finding interface files

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

`-i<dirs>`

> This flag appends a colon-separated list of `dirs` to the "import directories" list, which initially contains a single entry: ".".
>
> This list is scanned before any package directories (see Section 4.10) when looking for imports, but note that if you have a home module with the same name as a package module then this is likely to cause trouble in other ways, with link errors being the least nasty thing that can go wrong...
>
> See also Section 4.9.4 for the significance of using relative and absolute pathnames in the `-i` list.

`-i`

> resets the "import directories" list back to nothing.

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

4.9.3. Other options related to interface files

`-ddump-hi`

> Dumps the new interface to standard output.

`-ddump-hi-diffs`

The compiler does not overwrite an existing `.hi` interface file if the new one is the same as the old one; this is friendly to **make**. When an interface does change, it is often enlightening to be informed. The `-ddump-hi-diffs` option will make GHC run **diff** on the old and new `.hi` files.

`-ddump-minimal-imports`

Dump to the file "M.imports" (where M is the module being compiled) a "minimal" set of import declarations. You can safely replace all the import declarations in "M.hs" with those found in "M.imports". Why would you want to do that? Because the "minimal" imports (a) import everything explicitly, by name, and (b) import nothing that is not required. It can be quite painful to maintain this property by hand, so this flag is intended to reduce the labour.

4.9.4. The recompilation checker

`-no-recomp`

Turn off recompilation checking (which is on by default). Recompilation checking normally stops compilation early, leaving an existing `.o` file in place, if it can be determined that the module does not need to be recompiled.

In the olden days, GHC compared the newly-generated `.hi` file with the previous version; if they were identical, it left the old one alone and didn't change its modification date. In consequence, importers of a module with an unchanged output `.hi` file were not recompiled.

This doesn't work any more. Suppose module `C` imports module `B`, and `B` imports module `A`. So changes to `A.hi` should force a recompilation of `C`. And some changes to `A` (changing the definition of a function that appears in an inlining of a function exported by `B`, say) may conceivably not change `B.hi` one jot. So now…

GHC keeps a version number on each interface file, and on each type signature within the interface file. It also keeps in every interface file a list of the version numbers of everything it used when it last compiled the file. If the source file's modification date is earlier than the `.o` file's date (i.e. the source hasn't changed since the file was last compiled), and the reompilation checking is on, GHC will be clever. It compares the version numbers on the things it needs this time with the version numbers on the things it needed last time (gleaned from the interface file of the module being compiled); if they are all the same it stops compiling rather early in the process saying "Compilation IS NOT required". What a beautiful sight!

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

4.9.5. 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.

4.9.5.1. Dependency generation

Putting inter-dependencies of the form `Foo.o : Bar.hi` into your `Makefile` by hand is rather error-prone. Don't worry, GHC has support for automatically generating the required dependencies. Add the following to your `Makefile`:

```
depend :
        ghc -M $(HC_OPTS) $(SRCS)
```

Now, before you start compiling, and any time you change the `imports` in your program, do **make depend** before you do **make cool_pgm**. **ghc -M** will append the needed dependencies to your `Makefile`.

In general, if module `A` contains the line

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

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

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

If module `A` contains the line

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

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

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

(See Section 4.9.6 for details of `hi-boot` style interface files.) If `A` imports multiple modules, then there will be multiple lines with `A.o` as the target.

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

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

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

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

The options which affect dependency generation are:

`-w`

Turn off warnings about interface file shadowing.

`-f file`

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

`-o <osuf>`

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

`-s <suf>`

Make extra dependencies that declare that files with suffix `.<suf>_<osuf>` depend on interface files with suffix `.<suf>_hi`, or (for `{-# SOURCE #-}` imports) on `.hi-boot`. Multiple `-s` flags are permitted. For example, `-o hc -s a -s b` will make dependencies for `.hc` on `.hi`, `.a_hc` on `.a_hi`, and `.b_hc` on `.b_hi`. (Useful in conjunction with NoFib "ways".)

`--exclude-module=<file>`

Regard `<file>` as "stable"; i.e., exclude it from having dependencies on it.

`-x`

same as `--exclude-module`

`--exclude-directory=<dirs>`

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

`--include-module=<file>`

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

`--include-prelude`

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

4.9.6. 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 0 where
__export A TA{MkTA} ;
1 newtype TA = MkTA PrelBase.Int ;
```

The syntax is essentially the same as a normal `.hi` file (unfortunately), so 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. In a normal interface file, the "0" is the version number of the compiler which generated the interface file; it is used to ensure that we don't mix-and-match interface files between compiler versions. Leaving it as zero in an `hi-boot` file turns off this check.

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 0 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.

4.9.7. Orphan modules and instance declarations

Haskell specifies that when compiling module M, any instance declaration in any module "below" M is visible. (Module A is "below" M if A is imported directly by M, or if A is below a module that M imports directly.) In principle, GHC must therefore read the interface files of every module below M, just in case they contain an instance declaration that matters to M. This would be a disaster in practice, so GHC tries to be clever.

In particular, if an instance declaration is in the same module as the definition of any type or class mentioned in the head of the instance declaration, then GHC has to visit that interface file anyway. Example:

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

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

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

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

Here, neither D nor T is declared in module Orphan. We call such modules "orphan modules", defined thus:

- An *orphan module* contains at least one *orphan instance* or at least one *orphan rule*.

- An instance declaration in a module M is an *orphan instance* if none of the type constructors or classes mentioned in the instance head (the part after the "=>") are declared in M.

  Only the instance head counts. In the example above, it is not good enough for C's declaration to be in module A; it must be the declaration of D or T.

- A rewrite rule in a module M is an *orphan rule* if none of the variables, type constructors, or classes that are free in the left hand side of the rule are declared in M.

GHC identifies orphan modules, and visits the interface file of every orphan module below the module being compiled. This is usually wasted work, but there is no avoiding it. You should therefore do your best to have as few orphan modules as possible.

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

4.10. Packages

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

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

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

4.10.1. Using a package

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

`-package lib`

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

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

4.10.2. Maintaining a local set of packages

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

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

`-package-conf` *`file`*

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

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

4.10.3. Building a package from Haskell source

It takes some special considerations to build a new package:

- A package may contain several Haskell modules. A package may span many directories, or many packages may exist in a single directory. Packages may not be mutually recursive.
- A package has a name (e.g. `std`)
- The Haskell code in a package may be built into one or more Unix libraries (e.g. `libHSfoo.a`), or a single DLL on Windows (e.g. `HSfoo.dll`). The restriction to a single DLL on Windows is that the package system is used to tell the compiler when it should make an inter-DLL call rather than an intra-DLL call (inter-DLL calls require an extra indirection). *Building packages as DLLs doesn't work at the moment; see Section 11.3 for the gory details.*

  Versions of the Haskell libraries for use with GHCi may also be included: GHCi cannot load `.a` files directly, instead it will look for an object file called `HSfoo.o` (the object suffix varies between platforms, as usual) and load that. An object file can be built from a `.a` archive as follows (using GNU **ld** on Unix):

  ```
  ld -r --whole-archive -o HSfoo.o libHSfoo.a
  ```
- GHC does not maintain detailed cross-package dependency information. It does remember which modules in other packages the current module depends on, but not which things within those imported things.

To compile a module which is to be part of a new package, use the `-package-name` option:

`-package-name` *`foo`*

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

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

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

4.10.4. Package management

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

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

`--add-package`
`-a`

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

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

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

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

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

`--list-packages`
`-l`

This option displays the list of currently installed packages.

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

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

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

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

Removes the specified package from the installed configuration.

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

A package specification looks like this:

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

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

`name`

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

`import_dirs`

A list of directories containing interface files (`.hi` files) for this package.

`source_dirs`

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

`library_dirs`

A list of directories containing libraries for this package.

`hs_libraries`

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

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

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

`libHSfoo.a`

The name of the library on Unix systems.

`HSfoo.dll`

The name of the dynamic library on Windows systems.

`HSfoo.o`
`HSfoo.obj`

The object version of the library used by GHCi.

`extra_libraries`

A list of extra libraries for this package. The difference between `hs_libraries` and `extra_libraries` is that `hs_libraries` normally have several versions, to support profiling, parallel and other build options. The various versions are given different suffixes to distinguish them, for example the profiling version of the standard prelude library is named `libHSstd_p.a`, with the `_p` indicating that this is a profiling version. The suffix is added automatically by GHC for `hs_libraries` only, no suffix is added for libraries in `extra_libraries`.

Also, `extra_libraries` are placed on the linker command line after the `hs_libraries` for the same package. If your package has dependencies in the other direction (i.e. `extra_libraries` depends on `hs_libraries`), and the libraries are static, you might need to make two separate packages.

`include_dirs`

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

`c_includes`

A list of files to include for via-C compilations using this package. Typically this include file will contain function prototypes for any C functions used in the package, in case they end up being called as a result of Haskell functions from the package being inlined.

`package_deps`

A list of packages which this package depends on.

`extra_ghc_opts`

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

`extra_cc_opts`

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

`extra_ld_opts`

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

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

4.11. Optimisation (code improvement)

The `-O*` options specify convenient "packages" of optimisation flags; the `-f*` options described later on specify *individual* optimisations to be turned on/off; the `-m*` options specify *machine-specific* optimisations to be turned on/off.

4.11.1. `-O*`: convenient "packages" of optimisation flags.

There are *many* options that affect the quality of code produced by GHC. Most people only have a general goal, something like "Compile quickly" or "Make my program run like greased lightning." The following "packages" of optimisations (or lack thereof) should suffice.

Once you choose a `-O*` "package," stick with it—don't chop and change. Modules' interfaces *will* change with a shift to a new `-O*` option, and you may have to recompile a large chunk of all importing modules before your program can again be run safely (see Section 4.9.4).

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**

`-O0`:

Means "turn off all optimisation", reverting to the same settings as if no `-O` options had been specified. Saying `-O0` can be useful if eg. **make** has inserted a `-O` on the command line already.

`-O` or `-O1`:

Means: "Generate good-quality code without taking too long about it." Thus, for example: **ghc -c -O Main.lhs**

`-O2`:

Means: "Apply every non-dangerous optimisation, even if it means significantly longer compile times."

The avoided "dangerous" optimisations are those that can make runtime or space *worse* if you're unlucky. They are normally turned on or off individually.

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

`-Ofile <file>`:

(NOTE: not supported yet in GHC 5.x. Please ask if you're interested in this.)

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

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

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

We don't use a `-O*` flag for day-to-day work. We use `-O` to get respectable speed; e.g., when we want to measure something. When we want to go for broke, we tend to use `-O -fvia-C` (and we go for lots of coffee breaks).

The easiest way to see what `-O` (etc.) "really mean" is to run with `-v`, then stand back in amazement.

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

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

`-fexcess-precision`:

When this option is given, intermediate floating point values can have a *greater* precision/range than the final type. Generally this is a good thing, but some programs may rely on the exact precision/range of `Float`/`Double` values and should not use this option for their compilation.

`-fignore-asserts`:

Causes GHC to ignore uses of the function `Exception.assert` in source code (in other words, rewriting `Exception.assert p e` to `e` (see Section 7.13). This flag is turned on by `-O`.

`-fno-strictness`

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

`-fno-cpr-analyse`

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

`-funbox-strict-fields`:

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

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

will create a constructor `T` containing two unboxed floats if the `-funbox-strict-fields` flag is given. This may not always be an optimisation: if the `T` constructor is scrutinised and the floats passed to a non-strict function for example, they will have to be reboxed (this is done automatically by the compiler).

This option should only be used in conjunction with `-O`, in order to expose unfoldings to the compiler so the reboxing can be removed as often as possible. For example:

```
f :: T -> Float
f (T f1 f2) = f1 + f2
```

The compiler will avoid reboxing `f1` and `f2` by inlining `+` on floats, but only when `-O` is on.

Any single-constructor data is eligible for unpacking; for example

```
data T = T !(Int,Int)
```

will store the two `Int`s directly in the `T` constructor, by flattening the pair. Multi-level unpacking is also supported:

```
data T = T !S
data S = S !Int !Int
```

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

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

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

```
x = plusInt a b
```

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

-funfolding-creation-threshold<n>:

(Default: 45) Governs the maximum size that GHC will allow a function unfolding to be. (An unfolding has a "size" that reflects the cost in terms of "code bloat" of expanding that unfolding at at a call site. A bigger function would be assigned a bigger cost.)

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

Increasing this figure is more likely to result in longer compile times than faster code. The next option is more useful:

-funfolding-use-threshold<n>:

(Default: 8) This is the magic cut-off figure for unfolding: below this size, a function definition will be unfolded at the call-site, any bigger and it won't. The size computed for a function depends on two things: the actual size of the expression minus any discounts that apply (see `-funfolding-con-discount`).

4.12. Options related to a particular phase

4.12.1. Replacing the program for one or more phases

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

`-pgmL` *cmd*

Use *cmd* as the literate pre-processor.

`-pgmP` *cmd*

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

`-pgmc` *cmd*

Use *cmd* as the C compiler.

`-pgma` *cmd*

Use *cmd* as the assembler.

`-pgml` *`cmd`*

Use *`cmd`* as the linker.

`-pgmdll` *`cmd`*

Use *`cmd`* as the DLL generator.

`-pgmdep` *`cmd`*

Use *`cmd`* as the dependency generator.

4.12.2. Forcing options to a particular phase

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

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

GHC is itself a Haskell program, so if you need to pass options directly to GHC's runtime system you can enclose them in `+RTS ... -RTS` (see Section 4.16).

4.12.3. Options affecting the C pre-processor

`-cpp`

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

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

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

`-U`*`symbol`*

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

`-I`*`dir`*

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

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

`__HASKELL98__`

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

`__HASKELL__=98`

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

`__HASKELL1__`

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

`__GLASGOW_HASKELL__`

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

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

(For reference: the comparable symbols for other systems are: `__HUGS__` for Hugs, `__NHC__` for nhc98, and `__HBC__` for Chalmers.)

NB. This macro is set when pre-processing both Haskell source and C source, including the C source generated from a Haskell module (i.e. `.hs`, `.lhs`, `.c` and `.hc` files).

`__CONCURRENT_HASKELL__`

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

`__PARALLEL_HASKELL__`

Only defined when `-parallel` is in use! This symbol is defined when pre-processing Haskell (input) and pre-processing C (GHC output).

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

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

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

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

4.12.4. Options affecting the C compiler (if applicable)

If you are compiling with lots of foreign calls, you may need to tell the C compiler about some `#include` files. There is no real pretty way to do this, but you can use this hack from the command-line:

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

4.12.5. Options affecting code generation

`-fasm`

Use GHC's native code generator rather than compiling via C. This will compile faster (up to twice as fast), but may produce code that is slightly slower than compiling via C. `-fasm` is the default when optimisation is off (see Section 4.11).

`-fvia-C`

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

`-fno-code`

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

4.12.6. Options affecting linking

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

`-l`*`lib`*

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

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

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

`-package name`

If you are using a Haskell "package" (see Section 4.10), don't forget to add the relevant `-package` option when linking the program too: it will cause the appropriate libraries to be linked in with the program. Forgetting the `-package` option will likely result in several pages of link errors.

`-Ldir`

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

`-split-objs`

Tell the linker to split the single object file that would normally be generated into multiple object files, one per top-level Haskell function or type in the module. We use this feature for building GHC's libraries libraries (warning: don't use it unless you know what you're doing!).

`-static`

Tell the linker to avoid shared Haskell libraries, if possible. This is the default.

`-dynamic`

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

`-no-hs-main`

In the event you want to include ghc-compiled code as part of another (non-Haskell) program, the RTS will not be supplying its definition of `main()` at link-time, you will have to. To signal that to the driver script when linking, use `-no-hs-main`.

Notice that since the command-line passed to the linker is rather involved, you probably want to use **ghc** to do the final link of your 'mixed-language' application. This is not a requirement though, just try linking once with `-v` on to see what options the driver passes through to the linker.

4.13. Using Concurrent Haskell

GHC (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 4.14.4.

Concurrent Haskell is described in more detail in Chapter 3 in *Haskell Libraries*.

4.14. Using Parallel Haskell

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

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

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

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

4.14.1. Dummy's guide to using PVM

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

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

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

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

```
pvm $PVM_HOST_FILE
```

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

Control-D	exit **pvm, leaving it running**
halt	kill off this "parallel machine" & exit
add <host>	add **<host> as a processor**

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

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

4.14.2. Parallelism profiles

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

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

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

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

4.14.3. Other useful info about running parallel programs

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

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

4.14.4. RTS options for Concurrent/Parallel Haskell

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

-qp<N>:

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

-C[<us>]:

Sets the context switch interval to <s> seconds. A context switch will occur at the next heap block allocation after the timer expires (a heap block allocation occurs every 4k of allocation). With -C0 or -C, context switches will occur as often as possible (at every heap block allocation). By default, context switches occur every 20ms milliseconds. Note that GHC's internal timer ticks every 20ms, and the context switch timer is always a multiple of this timer, so 20ms is the maximum granularity available for timed context switches.

-q[v]:

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

-qt<num>:

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

-qe<num>:

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

-qQ<num>:

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

-qh<num>:

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

`-qg<num>`:

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

4.15. Platform-specific Flags

Some flags only make sense for particular target platforms.

`-mv8`:

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

`-monly-[32]-regs`:

(iX86 machines) GHC tries to “steal” four registers from GCC, for performance reasons; it almost always works. However, when GCC is compiling some modules with four stolen registers, it will crash, probably saying:

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

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

4.16. Running a compiled program

To make an executable program, the GHC system compiles your code and then links it with a non-trivial runtime system (RTS), which handles storage management, profiling, etc.

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

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

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

The RTS will snaffle `-p -S` for itself, and the remaining arguments `-f -h foo bar` will be handed to your program if/when it calls `System.getArgs`.

No `-RTS` option is required if the runtime-system options extend to the end of the command line, as in this example:

```
% hls -ltr /usr/etc +RTS -A5m
```

If you absolutely positively want all the rest of the options in a command line to go to the program (and not the RTS), use a `--RTS`.

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

Giving a `+RTS -f` option will print out the RTS options actually available in your program (which vary, depending on how you compiled).

NOTE: since GHC is itself compiled by GHC, you can change RTS options in the compiler using the normal `+RTS ... -RTS` combination. eg. to increase the maximum heap size for a compilation to 128M, you would add `+RTS -M128m -RTS` to the command line.

4.16.1. Setting global RTS options

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

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

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

4.16.2. RTS options to control the garbage collector

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

`-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).

`-c`

Use a compacting algorithm for collecting the oldest generation. By default, the oldest generation is collected using a copying algorithm; this option causes it to be compacted in-place instead. The compaction algorithm is slower than the copying algorithm, but the savings in memory use can be considerable.

For a given heap size (using the `-H` option), compaction can in fact reduce the GC cost by allowing fewer GCs to be performed. This is more likely when the ratio of live data to heap size is high, say >30%.

NOTE: compaction doesn't currently work when a single generation is requested using the `-G1` option.

`-cn`

[Default: 30] Automatically enable compacting collection when the live data exceeds *n*% of the maximum heap size (see the `-M` option). Note that the maximum heap size is unlimited by default, so this option has no effect unless the maximum heap size is set with `-Msize`.

`-Ffactor`

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

The default seems to work well here. If you have plenty of memory, it is usually better to use `-Hsize` than to increase `-Ffactor`.

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

`-Ggenerations`

[Default: 2] Set the number of generations used by the garbage collector. The default of 2 seems to be good, but the garbage collector can support any number of generations. Anything larger than about 4 is probably not a good idea unless your program runs for a *long* time, because the oldest generation will hardly ever get collected.

Specifying 1 generation with `+RTS -G1` gives you a simple 2-space collector, as you would expect. In a 2-space collector, the `-A` option (see above) specifies the *minimum* allocation area size, since the allocation area will grow with the amount of live data in the heap. In a multi-generational collector the allocation area is a fixed size (unless you use the `-H` option, see below).

`-Hsize`

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

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

`-ksize`

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

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

`-Ksize`

[Default: 1M] Set the maximum stack size for an individual thread to *size* bytes. This option is there purely to stop the program eating up all the available memory in the machine if it gets into an infinite loop.

`-mn`

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

`-Msize`

[Default: unlimited] Set the maximum heap size to *size* bytes. The heap normally grows and shrinks according to the memory requirements of the program. The only reason for having this option is to stop the heap growing without bound and filling up all the available swap space, which at the least will result in the program being summarily killed by the operating system.

The maximum heap size also affects other garbage collection parameters: when the amount of live data in the heap exceeds a certain fraction of the maximum heap size, compacting collection will be automatically enabled for the oldest generation, and the `-F` parameter will be reduced in order to avoid exceeding the maximum heap size.

`-sfile`
`-Sfile`

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

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

`-t`

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

4.16.3. RTS options for hackers, debuggers, and over-interested souls

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

`-B`

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

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

`-D`*`num`*

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

`-r`*`file`*

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

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

`-xc`

(Only available when the program is compiled for profiling.) When an exception is raised in the program, this option causes the current cost-centre-stack to be dumped to `stderr`.

This can be particularly useful for debugging: if your program is complaining about a `head []` error and you haven't got a clue which bit of code is causing it, compiling with `-prof -auto-all` and running with `+RTS -xc -RTS` will tell you exactly the call stack at the point the error was raised.

`-Z`

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

4.16.4. "Hooks" to change RTS behaviour

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

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

The function `defaultsHook` lets you change various RTS options. The commonest use for this is to give your program a default heap and/or stack size that is greater than the default. For example, to set `-M128m -K1m`:

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

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

You can also change the messages printed when the runtime system "blows up," e.g., on stack overflow. The hooks for these are as follows:

`void ErrorHdrHook (FILE *)`

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

`void OutOfHeapHook (unsigned long, unsigned long)`

The heap-overflow message.

`void StackOverflowHook (long int)`

The stack-overflow message.

`void MallocFailHook (long int)`

The message printed if `malloc` fails.

`void PatErrorHdrHook (FILE *)`

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

`void PreTraceHook (FILE *)`

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

`void PostTraceHook (FILE *)`

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

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

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

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

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

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

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

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

4.17. Generating External Core Files

GHC can dump its optimized intermediate code (said to be in "Core" format) to a file as a side-effect of compilation. Core files, which are given the suffix `.hcr`, can be read and processed by non-GHC back-end tools. The Core format is formally described in *An External Representation for the GHC Core Language* (http://www.haskell.org/ghc/docs/papers/core.ps.gz), and sample tools (in Haskell) for manipulating Core files are available in the GHC source distribution directory `/fptools/ghc/utils/ext-core`. Note that the format of `.hcr` files is *different* (though similar) to the Core output format generated for debugging purposes (Section 4.18).

`-fext-core`

Generate `.hcr` files.

4.18. Debugging the compiler

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

4.18.1. Dumping out compiler intermediate structures

`-ddump-`*`pass`*

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

`-ddump-parsed`:

parser output

`-ddump-rn`:

renamer output

`-ddump-tc`:

typechecker output

`-ddump-types`:

Dump a type signature for each value defined at the top level of the module. The list is sorted alphabetically. Using `-dppr-debug` dumps a type signature for all the imported and system-defined things as well; useful for debugging the compiler.

`-ddump-deriv`:

derived instances

`-ddump-ds`:

desugarer output

`-ddump-spec`:

output of specialisation pass

`-ddump-rules`:

dumps all rewrite rules (including those generated by the specialisation pass)

`-ddump-simpl`:

simplifer output (Core-to-Core passes)

`-ddump-inlinings`:

inlining info from the simplifier

`-ddump-usagesp`:

UsageSP inference pre-inf and output

`-ddump-cpranal`:

CPR analyser output

`-ddump-stranal`:

strictness analyser output

`-ddump-cse`:

CSE pass output

`-ddump-workwrap`:

worker/wrapper split output

`-ddump-occur-anal`:

'occurrence analysis' output

`-ddump-sat`:

output of "saturate" pass

`-ddump-stg`:

output of STG-to-STG passes

`-ddump-absC`:

*un*flattened Abstract C

`-ddump-flatC`:

flattened Abstract C

`-ddump-realC`:

same as what goes to the C compiler

`-ddump-stix`:

native-code generator intermediate form

`-ddump-asm`:

assembly language from the native-code generator

`-ddump-bcos`:

byte code compiler output

`-ddump-foreign`:

dump foreign export stubs

`-dverbose-core2core`
`-dverbose-stg2stg`

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

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

`-ddump-simpl-iterations`:

Show the output of each *iteration* of the simplifier (each run of the simplifier has a maximum number of iterations, normally 4). Used when even `-dverbose-simpl` doesn't cut it.

`-dppr-debug`

Debugging output is in one of several "styles." Take the printing of types, for example. In the "user" style (the default), the compiler's internal ideas about types are presented in Haskell source-level syntax, insofar as possible. In the "debug" style (which is the default for debugging output), the types are printed in with explicit foralls, and variables have their unique-id attached (so you can check for things that look the same but aren't). This flag makes debugging output appear in the more verbose debug style.

`-dppr-user-length`

In error messages, expressions are printed to a certain "depth", with subexpressions beyond the depth replaced by ellipses. This flag sets the depth.

`-ddump-simpl-stats`

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

`-ddump-rn-trace`

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

`-ddump-rn-stats`

Print out summary of what kind of information the renamer had to bring in.

`-dshow-unused-imports`

Have the renamer report what imports does not contribute.

4.18.2. Checking for consistency

`-dcore-lint`

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

`-dstg-lint`:

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

`-dusagesp-lint`:

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

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

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

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

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

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

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

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

_NI_

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

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

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

-# The '+' class method (Unique: r3JH) selects the addition code
-# from a 'Num' dictionary (now an explicit lamba'd argument).
```

```
-# Because Core is 2nd-order lambda-calculus, type applications
-# and lambdas (/\) are explicit.  So '+' is first applied to a
-# type ('_4'), then to a dictionary, yielding the actual addition
-# function that we will use subsequently...

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

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

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

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

          skip2.t3Ja :: _4 -> [_4]
          _NI_
          skip2.t3Ja =
              \ m.r1H4 ->
                  let { ds.d4QQ :: [_4]
                        _NI_
                        ds.d4QQ =
                    let {
                      ds.d4QY :: _4
                      _NI_
                      ds.d4QY = +.t4Hg m.r1H4 lit.t4Hb
                    } in  skip2.t3Ja ds.d4QY
                  } in
                  :! _4 m.r1H4 ds.d4QQ

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

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

4.18.4. Unregisterised compilation

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

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

`-unreg`:

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

4.19. Flag reference

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

4.19.1. Help and verbosity options (Section 4.4)

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

4.19.2. Which phases to run (Section 4.6)

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

4.19.3. Redirecting output (Section 4.7)

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

4.19.4. Keeping intermediate files (Section 4.7.1)

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

4.19.5. Temporary files (Section 4.7.2)

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

4.19.6. Finding imports (Section 4.9.2)

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

4.19.7. Interface file options (Section 4.9.3)

Flag	Description	Static/Dynamic	Reverse
`-ddump-hi`	Dump the new interface to stdout	dynamic	-
`-ddump-hi-diffs`	Show the differences vs. the old interface	dynamic	-

4.19.8. Recompilation checking (Section 4.9.4)

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

4.19.9. Interactive-mode options (Section 3.8)

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

4.19.10. Packages (Section 4.10)

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

4.19.11. Language options (Section 7.1)

Flag	Description	Static/Dynamic	Reverse
`-fallow-overlapping-instances`		dynamic	`-fno-allow-overlapping-instances`
`-fallow-undecidable-instances`	Enable undecidable instances	dynamic	`-fno-allow-undecidable-instances`
`-fgenerics`	Enable generics	dynamic	`-fno-fgenerics`
`-fglasgow-exts`	Enable most language extensions	dynamic	`-fno-glasgow-exts`
`-fignore-asserts`	Ignore assertions	dynamic	`-fno-ignore-asserts`
`-fno-implicit-prelude`	Don't implicitly `import Prelude`	dynamic	-
`-fno-monomorphism-restriction`	Disable the monomorphism restriction	dynamic	–
`-firrefutable-tuples`	Make tuple pattern matching irrefutable	dynamic	`-fno-irrefutable-tuples`
`-fcontext-stackn`	set the limit for context reduction	dynamic	–

4.19.12. Warnings (Section 4.8)

Flag	Description	Static/Dynamic	Reverse
`-W`	enable normal warnings	dynamic	`-w`
`-w`	disable all warnings	dynamic	-
`-Wall`	enable all warnings	dynamic	`-w`
`-fwarn-deprecations`	warn about uses of functions & types that are deprecated	dynamic	`-fno-warn-deprecations`
`-fwarn-duplicate-exports`	warn when an entity is exported multiple times	dynamic	`-fno-warn-duplicate-exports`
`-fwarn-hi-shadowing`	warn when a `.hi file in the current directory shadows a library`	dynamic	`-fno-warn-hi-shadowing`

Flag	Description	Static/Dynamic	Reverse
`-fwarn-incomplete-patterns`	warn when a pattern match could fail	dynamic	`-fno-warn-incomplete-patterns`
`-fwarn-misc`	enable miscellaneous warnings	dynamic	`-fno-warn-misc`
`-fwarn-missing-fields`	warn when fields of a record are uninitialised	dynamic	`-fno-warn-missing-fields`
`-fwarn-missing-methods`	warn when class methods are undefined	dynamic	`-fno-warn-missing-methods`
`-fwarn-missing-signatures`	warn about top-level functions without signatures	dynamic	`-fno-warn-missing-signatures`
`-fwarn-name-shadowing`	warn when names are shadowed	dynamic	`-fno-warn-name-shadowing`
`-fwarn-overlapping-patterns`	warn about overlapping patterns	dynamic	`-fno-warn-overlapping-patterns`
`-fwarn-simple-patterns`	warn about lambda-patterns that can fail	dynamic	`-fno-warn-simple-patterns`
`-fwarn-type-defaults`	warn when defaulting happens	dynamic	`-fno-warn-type-defaults`
`-fwarn-unused-binds`	warn about bindings that are unused	dynamic	`-fno-warn-unused-binds`
`-fwarn-unused-imports`	warn about unnecessary imports	dynamic	`-fno-warn-unused-imports`
`-fwarn-unused-matches`	warn about variables in patterns that aren't used	dynamic	`-fno-warn-unused-matches`

4.19.13. Optimisation levels (Section 4.11)

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

4.19.14. Individual optimisations (Section 4.11.2)

Flag	Description	Static/Dynamic	Reverse
`-fall-strict`	Make everything strict	dynamic	`-fno-all-strict`
`-fcase-merge`	Enable case-merging	static	`-fno-case-merge`
`-fdicts-strict`	Make dictionaries strict	dynamic	`-fno-dicts-strict`
`-fdo-eta-reduction`	Enable eta-reduction	static	`-fno-do-eta-reduction`

Flag	Description	Static/Dynamic	Reverse
`-fdo-lambda-eta-expansion`	Enable lambda eta-reduction	static	`-fno-do-lambda-eta-expansion`
`-fexcess-precision`	Enable excess intermediate precision	static	`-fno-excess-precision`
`-ffoldr-build-on`	Enable foldr-build optimisation	static	`-fno-foldr-build-on`
`-fignore-asserts`	Ignore assertions in the source	static	`-fno-ignore-asserts`
`-fignore-interface-pragmas`	Ignore pragmas in interface files	static	`-fno-ignore-interface-pragmas`
`-flet-no-escape`	Enable let-no-escape optimisation	static	`-fno-let-no-escape`
`-fliberate-case-threshold`	Tweak the liberate-case optimisation (default: 10)	static	`-fno-liberate-case-threshold`
`-fomit-interface-pragmas`	Don't generate interface pragmas	static	`-fno-omit-interface-pragmas`
`-fmax-worker-args`	If a worker has that many arguments, none will be unpacked anymore (default: 10)	static	-
`-fmax-simplifier-iterations`	Set the max iterations for the simplifier	static	-
`-fno-cpr`	Turn off CPR analysis	static	-
`-fno-cse`	Turn off common sub-expression	static	-
`-fno-pre-inlining`	Turn off pre-inlining	static	-
`-fno-strictness`	Turn off strictness analysis	static	-
`-fnumbers-strict`	Make numbers strict	dynamic	`-fno-numbers-strict`
`-funbox-strict-fields`	Flatten strict constructor fields	static	`-fno-unbox-strict-fields`
`-funfolding-creation-threshold`	Tweak unfolding settings	static	`-fno-unfolding-creation-threshold`
`-funfolding-fun-discount`	Tweak unfolding settings	static	`-fno-unfolding-fun-discount`
`-funfolding-keeness-factor`	Tweak unfolding settings	static	`-fno-unfolding-keeness-factor`
`-funfolding-update-in-place`	Tweak unfolding settings	static	`-fno-unfolding-update-in-place`
`-funfolding-use-threshold`	Tweak unfolding settings	static	`-fno-unfolding-use-threshold`
`-fusagesp`	Turn on UsageSP analysis	static	`-fno-usagesp`

4.19.15. Profiling options (Chapter 5)

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

4.19.16. Parallelism options (Section 4.14)

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

4.19.17. C pre-processor options (Section 4.12.3)

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

4.19.18. C compiler options (Section 4.12.4)

Flag	Description	Static/Dynamic	Reverse

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

4.19.19. Code generation options (Section 4.12.5)

Flag	Description	Static/Dynamic	Reverse
`-fasm`	Use the native code generator	dynamic	-fvia-C
`-fvia-C`	Compile via C	dynamic	-fasm
`-fno-code`	Omit code generation	static	-

4.19.20. Linking options (Section 4.12.6)

Flag	Description	Static/Dynamic	Reverse
`-dynamic`	Use dynamic Haskell libraries (if available)	static	-
`-llib`	Link in library *lib*	static	-
`-Ldir`	Add *dir to the list of directories searched for libraries*	static	-
`-package name`	Link in package *name*	static	-
`-split-objs`	Split objects (for libraries)	static	-
`-static`	Use static Haskell libraries	static	-
`-no-hs-main`	Don't asssume this program contains `main`	static	-

4.19.21. Replacing phases (Section 4.12.1)

Flag	Description	Static/Dynamic	Reverse
`-pgmL cmd`	Use *cmd as the literate pre-processor*	static	-
`-pgmP cmd`	Use *cmd as the C pre-processor (with -cpp only)*	static	-

Flag	Description	Static/Dynamic	Reverse
-pgmc *cmd*	Use *cmd as the C compiler*	static	-
-pgma *cmd*	Use *cmd as the assembler*	static	-
-pgml *cmd*	Use *cmd as the linker*	static	-
-pgmdll *cmd*	Use *cmd as the DLL generator*	static	-
-pgmdep *cmd*	Use *cmd as the dependency generator*	static	-

4.19.22. Forcing options to particular phases (Section 4.12.2)

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

4.19.23. Platform-specific options (Section 4.15)

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

4.19.24. Compiler debugging options (Section 4.18)

Flag	Description	Static/Dynamic	Reverse
`-dcore-lint`	Turn on internal sanity checking	dynamic	-
`-ddump-absC`	Dump abstract C	dynamic	-
`-ddump-asm`	Dump assembly	dynamic	-
`-ddump-bcos`	Dump interpreter byte code	dynamic	-
`-ddump-cpranal`	Dump output from CPR analysis	dynamic	-
`-ddump-cse`	Dump CSE output	dynamic	-
`-ddump-deriv`	Dump deriving output	dynamic	-
`-ddump-ds`	Dump desugarer output	dynamic	-
`-ddump-flatC`	Dump “flat” C	dynamic	-
`-ddump-foreign`	Dump `foreign export stubs`	dynamic	-
`-ddump-inlinings`	Dump inlining info	dynamic	-
`-ddump-occur-anal`	Dump occurrence analysis output	dynamic	-
`-ddump-parsed`	Dump parse tree	dynamic	-
`-ddump-realC`	Dump “real” C	dynamic	-
`-ddump-rn`	Dump renamer output	dynamic	-
`-ddump-rules`	Dump rules	dynamic	-
`-ddump-sat`	Dump saturated output	dynamic	-
`-ddump-simpl`	Dump final simplifier output	dynamic	-
`-ddump-simpl-iterations`	Dump output from each simplifier iteration	dynamic	-
`-ddump-spec`	Dump specialiser output	dynamic	-
`-ddump-stg`	Dump final STG	dynamic	-
`-ddump-stranal`	Dump strictness analyser output	dynamic	-
`-ddump-tc`	Dump typechecker output	dynamic	-
`-ddump-types`	Dump type signatures	dynamic	-
`-ddump-usagesp`	Dump UsageSP analysis output	dynamic	-
`-ddump-worker-wrapper`	Dump worker-wrapper output	dynamic	-
`-ddump-rn-trace`	Trace renamer	dynamic	-
`-ddump-rn-stats`	Renamer stats	dynamic	-
`-ddump-stix`	Native code generator intermediate form	dynamic	-
`-ddump-simpl-stats`	Dump simplifier stats	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-dppr-debug`	Turn on debug printing (more verbose)	static	-
`-dppr-noprags`	Don't output pragma info in dumps	static	-
`-dppr-user-length`	Set the depth for printing expressions in error msgs	static	-
`-dsource-stats`	Dump haskell source stats	dynamic	-
`-dstg-lint`	STG pass sanity checking	dynamic	-
`-dstg-stats`	Dump STG stats	dynamic	-
`-dusagesp-lint`	UsageSP sanity checker	dynamic	-
`-dverbose-core2core`	Show output from each core-to-core pass	dynamic	-
`-dverbose-stg2stg`	Show output from each STG-to-STG pass	dynamic	-
`-unreg`	Enable unregisterised compilation	static	-

4.19.25. Misc compiler options

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

Flag	Description	Static/Dynamic	Reverse
`-fno-asm-mangling`	Turn off assembly mangling (use `-unreg instead)`	static	-

Chapter 5. Profiling

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

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

Profiling a program is a three-step process:

1. Re-compile your program for profiling with the `-prof` option, and probably one of the `-auto` or `-auto-all` options. These options are described in more detail in Section 5.4
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.

5.1. Cost centres and cost-centre stacks

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

Let's take a look at an example:

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

Compile and run this program as follows:

```
$ ghc -prof -auto-all -o Main Main.hs
$ ./Main +RTS -p
121393
$
```

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

```
          Fri May 12 14:06 2000 Time and Allocation Profiling Report  (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 5.3) will be able to display real loops in the call-graph.

5.1.1. Inserting cost centres by hand

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

The syntax of a cost centre annotation is

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

where `"name"` is an aribrary string, that will become the name of your cost centre as it appears in the profiling output, and `<expression>` is any Haskell expression. An `SCC` annotation extends as far to the right as possible when parsing.

5.1.2. Rules for attributing costs

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

- If the expression is part of the *one-off* costs of evaluating the enclosing top-level definition, then costs are attributed to the stack of lexically enclosing `SCC` annotations on top of the special `CAF` cost-centre.
- Otherwise, costs are attributed to the stack of lexically-enclosing `SCC` annotations, appended to the cost-centre stack in effect at the *call site* of the current top-level definition[1]. Notice that this is a recursive definition.

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

```
x = nfib 25
```

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

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

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

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

5.2. 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 5.5. Details on the usage of the **hp2ps** program are given in Section 5.6

5.3. 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.informatik.uni-bremen.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).

5.4. 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.

5.5. 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 5.3.

`-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 5.6).

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.

`-h<filtering-options>`:

It's often useful to select just some subset of the heap when profiling. To do this, the following filters are available. You may use multiple filters, in which case a closure has to satisfy all filters to appear in the final profile. Filtering criterion are independent of what it is you ask to see. So, for example, you can specify a profile by closure description (`-hD`) but ask to filter closures by producer module (`-hm{...}`).

Available filters are:

`-hc{cc1, cc2 .. ccN}`:

Restrict to one of the specified cost centers. Since GHC deals in cost center stacks, the specified cost centers pertain to the top stack element. For example, `-hc{Wurble,Burble}` selects all cost center stacks whose top element is `Wurble` or `Burble`.

`-hm{module1, module2 .. moduleN}`:

Restrict to closures produced by functions in one of the specified modules.

`-hd{descr1, descr2 .. descrN}:`

Restrict to closures whose description-string is one of the specified descriptions. Description strings are pretty arcane. An easy way to find plausible strings to specify is to first do a `-hD` profile and then inspect the description-strings which appear in the resulting profile.

`-hy{type1, type2 .. typeN}:`

Restrict to closures having one of the specified types.

`-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).

`-xc`

This option makes use of the extra information maintained by the cost-centre-stack profiler to provide useful information about the location of runtime errors. See Section 4.16.3.

5.6. hp2ps—heap profile to PostScript

Usage:

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

The program **hp2ps** converts a heap profile as produced by the `-h<break-down>` runtime option into a PostScript graph of the heap profile. By convention, the file to be processed by **hp2ps** has a `.hp` extension. The PostScript output is written to `<file>@.ps`. If `<file>` is omitted entirely, then the program behaves as a filter.

hp2ps is distributed in `ghc/utils/hp2ps` in a GHC source distribution. It was originally developed by Dave Wakeling as part of the HBC/LML heap profiler.

The flags are:

`-d`

In order to make graphs more readable, **hp2ps** sorts the shaded bands for each identifier. The default sort ordering is for the bands with the largest area to be stacked on top of the smaller ones. The `-d` option causes rougher bands (those representing series of values with the largest standard deviations) to be stacked on top of smoother ones.

`-b`

Normally, **hp2ps** puts the title of the graph in a small box at the top of the page. However, if the JOB string is too long to fit in a small box (more than 35 characters), then **hp2ps** will choose to use a big box instead. The `-b` option forces **hp2ps** to use a big box.

`-e<float>[in|mm|pt]`

Generate encapsulated PostScript suitable for inclusion in LaTeX documents. Usually, the PostScript graph is drawn in landscape mode in an area 9 inches wide by 6 inches high, and **hp2ps** arranges for this area to be approximately centred on a sheet of a4 paper. This format is convenient of studying the graph in detail, but it is unsuitable for inclusion in LaTeX documents. The `-e` option causes the graph to be drawn in portrait mode, with float specifying the width in inches, millimetres or points (the default). The resulting PostScript file conforms to the Encapsulated PostScript (EPS) convention, and it can be included in a LaTeX document using Rokicki's dvi-to-PostScript converter **dvips**.

`-g`

Create output suitable for the **gs** PostScript previewer (or similar). In this case the graph is printed in portrait mode without scaling. The output is unsuitable for a laser printer.

`-l`

Normally a profile is limited to 20 bands with additional identifiers being grouped into an `OTHER` band. The `-l` flag removes this 20 band and limit, producing as many bands as necessary. No key is produced as it won't fit!. It is useful for creation time profiles with many bands.

`-m<int>`

Normally a profile is limited to 20 bands with additional identifiers being grouped into an `OTHER` band. The `-m` flag specifies an alternative band limit (the maximum is 20).

`-m0` requests the band limit to be removed. As many bands as necessary are produced. However no key is produced as it won't fit! It is useful for displaying creation time profiles with many bands.

`-p`

Use previous parameters. By default, the PostScript graph is automatically scaled both horizontally and vertically so that it fills the page. However, when preparing a series of graphs for use in a presentation, it is often useful to draw a new graph using the same scale, shading and ordering as a previous one. The `-p` flag causes the graph to be drawn using the parameters determined by a previous run of **hp2ps** on `file`. These are extracted from `file@.aux`.

`-s`

Use a small box for the title.

`-t<float>`

Normally trace elements which sum to a total of less than 1% of the profile are removed from the profile. The `-t` option allows this percentage to be modified (maximum 5%).

`-t0` requests no trace elements to be removed from the profile, ensuring that all the data will be displayed.

`-c`

Generate colour output.

`-y`

Ignore marks.

`-?`

Print out usage information.

5.7. Using “ticky-ticky” profiling (for implementors)

(ToDo: document properly.)

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

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

To be able to use ticky-ticky profiling, you will need to have built appropriate libraries and things when you made the system. See “Customising what libraries to build,” in the installation guide.

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

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

```
 foo +RTS -rfoo.ticky


ALLOCATIONS: 3964631 (11330900 words total: 3999476 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      0
UPD_PAP_IN_NEW:     34825:       0      0      0  34825      0      0      0      0      0

NEW GEN UPDATES: 2274700 ( 99.9%)

OLD GEN UPDATES: 1852 (  0.1%)

Total bytes copied during GC: 190096
```

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

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

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

Notes

1. The call-site is just the place in the source code which mentions the particular function or variable.
2. daVinci is sadly not open-source :-(.

Chapter 6. Advice on: sooner, faster, smaller, stingier

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

6.1. Sooner: producing a program more quickly

Don’t use `-O` or (especially) `-O2`:

By using them, you are telling GHC that you are willing to suffer longer compilation times for better-quality code.

GHC is surprisingly zippy for normal compilations without `-O`!

Use more memory:

Within reason, more memory for heap space means less garbage collection for GHC, which means less compilation time. If you use the `-Rghc-timing` option, you’ll get a garbage-collector report. (Again, you can use the cheap-and-nasty `+RTS -Sstderr -RTS` option to send the GC stats straight to standard error.)

If it says you’re using more than 20% of total time in garbage collecting, then more memory would help.

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

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

If GHC persists in being a bad memory citizen, please report it as a bug.

Don’t use too much memory!

As soon as GHC plus its “fellow citizens” (other processes on your machine) start using more than the *real memory* on your machine, and the machine starts “thrashing,” *the party is over*. Compile times will be worse than terrible! Use something like the csh-builtin **time** command to get a report on how many page faults you’re getting.

If you don’t know what virtual memory, thrashing, and page faults are, or you don’t know the memory configuration of your machine, *don’t* try to be clever about memory use: you’ll just make your life a misery (and for other people, too, probably).

Try to use local disks when linking:

Because Haskell objects and libraries tend to be large, it can take many real seconds to slurp the bits to/from a remote filesystem.

It would be quite sensible to *compile* on a fast machine using remotely-mounted disks; then *link* on a slow machine that had your disks directly mounted.

Don't derive/use `Read` unnecessarily:

It's ugly and slow.

GHC compiles some program constructs slowly:

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

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

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

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

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

Explicit `import` declarations:

Instead of saying `import Foo`, say `import Foo (...stuff I want...)` You can get GHC to tell you the minimal set of required imports by using the `-ddump-minimal-imports` option (see Section 4.9.3).

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

6.2. Faster: producing a program that runs quicker

The key tool to use in making your Haskell program run faster are GHC's profiling facilities, described separately in Chapter 5. There is *no substitute* for finding where your program's time/space is *really* going, as opposed to where you imagine it is going.

Another point to bear in mind: By far the best way to improve a program's performance *dramatically* is to use better algorithms. Once profiling has thrown the spotlight on the guilty time-consumer(s), it may be better to re-think your program than to try all the tweaks listed below.

Another extremely efficient way to make your program snappy is to use library code that has been Seriously Tuned By Someone Else. You *might* be able to write a better quicksort than the one in the HBC library, but it will take you much longer than typing `import QSort`. (Incidentally, it doesn't hurt if the Someone Else is Lennart Augustsson.)

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

Optimise, using `-O` or `-O2`:

This is the most basic way to make your program go faster. Compilation time will be slower, especially with `-O2`.

At present, `-O2` is nearly indistinguishable from `-O`.

Compile via C and crank up GCC:

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

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

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

Overloaded functions are not your friend:

Haskell's overloading (using type classes) is elegant, neat, etc., etc., but it is death to performance if left to linger in an inner loop. How can you squash it?

Give explicit type signatures:

Signatures are the basic trick; putting them on exported, top-level functions is good software-engineering practice, anyway. (Tip: using `-fwarn-missing-signatures` can help enforce good signature-practice).

The automatic specialisation of overloaded functions (with `-O`) should take care of overloaded local and/or unexported functions.

Use `SPECIALIZE` pragmas:

Specialize the overloading on key functions in your program. See Section 7.15.3 and Section 7.15.4.

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

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

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

Strict functions are your dear friends:

and, among other things, lazy pattern-matching is your enemy.

(If you don't know what a "strict function" is, please consult a functional-programming textbook. A sentence or two of explanation here probably would not do much good.)

Consider these two code fragments:

```
f (Wibble x y) =  ... # strict

f arg = let { (Wibble x y) = arg } in ... # lazy
```

The former will result in far better code.

A less contrived example shows the use of `cases` instead of `lets` to get stricter code (a good thing):

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

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

GHC loves single-constructor data-types:

It's all the better if a function is strict in a single-constructor type (a type with only one data-constructor; for example, tuples are single-constructor types).

Newtypes are better than datatypes:

If your datatype has a single constructor with a single field, use a `newtype` declaration instead of a `data` declaration. The `newtype` will be optimised away in most cases.

"How do I find out a function's strictness?"

Don't guess—look it up.

Look for your function in the interface file, then for the third field in the pragma; it should say `__S <string>`. The `<string>` gives the strictness of the function's arguments. `L` is lazy (bad), `S` and `E` are strict (good), `P` is "primitive" (good), `U(...)` is strict and "unpackable" (very good), and `A` is absent (very good).

For an "unpackable" `U(...)` argument, the info inside tells the strictness of its components. So, if the argument is a pair, and it says `U(AU(LSS))`, that means "the first component of the pair isn't used; the second component is itself unpackable, with three components (lazy in the first, strict in the second \& third)."

If the function isn't exported, just compile with the extra flag `-ddump-simpl`; next to the signature for any binder, it will print the self-same pragmatic information as would be put in an interface file. (Besides, Core syntax is fun to look at!)

Force key functions to be `INLINE`d (esp. monads):

Placing `INLINE` pragmas on certain functions that are used a lot can have a dramatic effect. See Section 7.15.1.

Explicit `export` list:

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

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

Look at the Core syntax!

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

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

Use unboxed types (a GHC extension):

When you are *really* desperate for speed, and you want to get right down to the "raw bits." Please see Section 7.2.1 for some information about using unboxed types.

Use `foreign import` (a GHC extension) to plug into fast libraries:

This may take real work, but... There exist piles of massively-tuned library code, and the best thing is not to compete with it, but link with it.

Chapter 8 describes the foreign function interface.

Don't use `Float`s:

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

Use a bigger heap!

If your program's GC stats (`-S` RTS option) indicate that it's doing lots of garbage-collection (say, more than 20% of execution time), more memory might help—with the `-M<size>` or `-A<size>` RTS options (see Section 4.16.2).

6.3. Smaller: producing a program that is smaller

Decrease the "go-for-it" threshold for unfolding smallish expressions. Give a `-funfolding-use-threshold0` option for the extreme case. ("Only unfoldings with zero cost should proceed.") Warning: except in certain specialised cases (like Happy parsers) this is likely to actually *increase* the size of your program, because unfolding generally enables extra simplifying optimisations to be performed.

Avoid `Read`.

Use `strip` on your executables.

6.4. 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 5) are the basic tool for demystifying the space behaviour of your program.

Strict functions are good for space usage, as they are for time, as discussed in the previous section. Strict functions get right down to business, rather than filling up the heap with closures (the system's notes to itself about how to evaluate something, should it eventually be required).

Chapter 7. GHC Language Features

As with all known Haskell systems, GHC implements some extensions to the language. To use them, you'll need to give a `-fglasgow-exts` option.

Virtually all of the Glasgow extensions serve to give you access to the underlying facilities with which we implement Haskell. Thus, you can get at the Raw Iron, if you are willing to write some non-standard code at a more primitive level. You need not be "stuck" on performance because of the implementation costs of Haskell's "high-level" features—you can always code "under" them. In an extreme case, you can write all your time-critical code in C, and then just glue it together with Haskell!

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 7.2.1 and following.

Type system extensions:

> GHC supports a large number of extensions to Haskell's type system. Specifically:
>
> Multi-parameter type classes:
>
> > Section 7.8
>
> Functional dependencies:
>
> > Section 7.10
>
> Implicit parameters:
>
> > Section 7.9
>
> Local universal quantification:
>
> > Section 7.11
>
> Extistentially quantification in data types:
>
> > Section 7.12
>
> 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 7.14.

Pattern guards

Instead of being a boolean expression, a guard is a list of qualifiers, exactly as in a list comprehension. See Section 7.6.

Data types with no constructors

See Section 7.5.

Parallel list comprehensions

An extension to the list comprehension syntax to support `zipWith`-like functionality. See Section 7.7.

Foreign calling:

Just what it sounds like. We provide *lots* of rope that you can dangle around your neck. Please see Chapter 8.

Pragmas

Pragmas are special instructions to the compiler placed in the source file. The pragmas GHC supports are described in Section 7.15.

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 7.16.

Generic classes:

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

Generic class declarations allow you to define a class whose methods say how to work over an arbitrary data type. Then it's really easy to make any new type into an instance of the class. This generalises the rather ad-hoc "deriving" feature of Haskell 98. Details in Section 7.17.

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*.

7.1. Language options

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

`-fglasgow-exts`:

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

`-fno-monomorphism-restriction`:

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

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

See Section 7.8.3. Only relevant if you also use `-fglasgow-exts`.

`-finline-phase`

See Section 7.16. Only relevant if you also use `-fglasgow-exts`.

`-fgenerics`

See Section 7.17. Independent of `-fglasgow-exts`.

`-fno-implicit-prelude`

GHC normally imports `Prelude.hi` files for you. If you'd rather it didn't, then give it a `-fno-implicit-prelude` option. The idea is that you can then import a Prelude of your own. (But don't call it `Prelude`; the Haskell module namespace is flat, and you must not conflict with any Prelude module.)

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

With one group of exceptions! You may want to define your own numeric class hierarchy. It completely defeats that purpose if the literal "1" means "`Prelude.fromInteger 1`", which is what the Haskell Report specifies. So the `-fno-implicit-prelude` flag causes the following pieces of built-in syntax to refer to *whatever is in scope*, not the Prelude versions:

- Integer and fractional literals mean "`fromInteger 1`" and "`fromRational 3.2`", not the Prelude-qualified versions; both in expressions and in patterns.
- Negation (e.g. "`- (f x)`") means "`negate (f x)`" (not `Prelude.negate`).
- In an n+k pattern, the standard Prelude `Ord` class is still used for comparison, but the necessary subtraction uses whatever "`(-)`" is in scope (not "`Prelude.(-)`").

Note: Negative literals, such as `-3`, are specified by (a careful reading of) the Haskell Report as meaning `Prelude.negate (Prelude.fromInteger 3)`. However, GHC deviates from this slightly, and treats them as meaning `fromInteger (-3)`. One particular effect of this slightly-non-standard reading is that there is no difficulty with the literal `-2147483648` at type `Int`; it means `fromInteger (-2147483648)`. The strict interpretation would be `negate (fromInteger 2147483648)`, and the call to `fromInteger` would overflow (at type `Int`, remember).

7.2. Unboxed types and primitive operations

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

Note: while you really can use this stuff to write fast code, we generally find it a lot less painful, and more satisfying in the long run, to use higher-level language features and libraries. With any luck, the code you write will be optimised to the efficient unboxed version in any case. And if it isn't, we'd like to know about it.

7.2.1. Unboxed types

Most types in GHC are *boxed*, which means that values of that type are represented by a pointer to a heap object. The representation of a Haskell `Int`, for example, is a two-word heap object. An *unboxed* type, however, is represented by the value itself, no pointers or heap allocation are involved.

Unboxed types correspond to the "raw machine" types you would use in C: `Int#` (long int), `Double#` (double), `Addr#` (void *), etc. The *primitive operations* (PrimOps) on these types are what you might expect; e.g., `(+#)` is addition on `Int#`s, and is the machine-addition that we all know and love—usually one instruction.

Primitive (unboxed) types cannot be defined in Haskell, and are therefore built into the language and compiler. Primitive types are always unlifted; that is, a value of a primitive type cannot be bottom. We use the convention that primitive types, values, and operations have a `#` suffix.

Primitive values are often represented by a simple bit-pattern, such as `Int#`, `Float#`, `Double#`. But this is not necessarily the case: a primitive value might be represented by a pointer to a heap-allocated object. Examples include `Array#`, the type of primitive arrays. A primitive array is heap-allocated because it is too big a value to fit in a register, and would be too expensive to copy around; in a sense, it is accidental that it is represented by a pointer. If a pointer represents a primitive value, then it really does point to that value: no unevaluated thunks, no indirections... nothing can be at the other end of the pointer than the primitive value.

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

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

7.2.2. Unboxed Tuples

Unboxed tuples aren't really exported by `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.

7.2.3. 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
```

7.2.4. Comparison operations

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

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

7.2.5. Primitive-character operations

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

7.2.6. 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!

7.2.7. 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.)

7.2.8. 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.
```

7.2.9. 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".

7.2.10. 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 ith offset gets you the ith `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.

7.2.11. 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 7.2.14), and "indexing" to refer to reading a value from an *immutable* array.

Immutable byte arrays are straightforward to index (all indices are in units of the size of the object being read):

```
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 7.2.2).

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

7.2.12. 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
```

7.2.13. 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).

7.2.14. 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
```

7.2.14.1. 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.

7.2.14.2. 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
```

7.2.14.3. 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
```

7.2.14.4. 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# #)
```

7.2.15. 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
```

7.3. 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 5.31 in *Haskell Libraries*.

7.4. 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 5.34 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 5.14 in *Haskell Libraries* for more details.

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

7.5. Data types with no constructors

With the `-fglasgow-exts` flag, GHC lets you declare a data type with no constructors. For example:

```
data S      - S :: *
data T a    - T :: * -> *
```

Syntactically, the declaration lacks the "= constrs" part. The type can be parameterised, but only over ordinary types, of kind *; since Haskell does not have kind signatures, you cannot parameterise over higher-kinded types.

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

7.6. Pattern guards

The discussion that follows is an abbreviated version of Simon Peyton Jones's original proposal (http://research.microsoft.com/~simonpj/Haskell/guards.html). (Note that the proposal was written before pattern guards were implemented, so refers to them as unimplemented.)

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

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

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

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

The auxiliary functions are

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

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

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

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

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

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

Here is how I would write clunky:

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

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

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

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

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

7.7. Parallel List Comprehensions

Parallel list comprehensions are a natural extension to list comprehensions. List comprehensions can be thought of as a nice syntax for writing maps and filters. Parallel comprehensions extend this to include the zipWith family.

A parallel list comprehension has multiple independent branches of qualifier lists, each separated by a '|' symbol. For example, the following zips together two lists:

```
[ (x, y) | x <- xs | y <- ys ]
```

The behavior of parallel list comprehensions follows that of zip, in that the resulting list will have the same length as the shortest branch.

We can define parallel list comprehensions by translation to regular comprehensions. Here's the basic idea:

Given a parallel comprehension of the form:

```
[ e | p1 <- e11, p2 <- e12, ...
    | q1 <- e21, q2 <- e22, ...
    ...
]
```

This will be translated to:

```
[ e | ((p1,p2), (q1,q2), ...) <- zipN [(p1,p2) | p1 <- e11, p2 <-
e12, ...]
                                      [(q1,q2) | q1 <- e21, q2 <-
e22, ...]
                                      ...
]
```

where 'zipN' is the appropriate zip for the given number of branches.

7.8. Multi-parameter type classes

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

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

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

7.8.1. Types

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

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

(Here, I write the "foralls" explicitly, although the Haskell source language omits them; in Haskell 1.4, all the free type variables of an explicit source-language type signature are universally quantified, except for the class type variables in a class declaration. However, in GHC, you can give the foralls if you want. See Section 7.11).

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

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

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

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

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

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

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

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

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

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

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

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

7.8.2. Class declarations

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

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

2. *The class hierarchy must be acyclic*. However, the definition of "acyclic" involves only the superclass relationships. For example, this is OK:

```
class C a where {
  op :: D b => a -> b -> b
}

class C a => D a where { ... }
```

Here, `C` is a superclass of `D`, but it's OK for a class operation `op` of `C` to mention `D`. (It would not be OK for `D` to be a superclass of `C`.)

3. *There are no restrictions on the context in a class declaration (which introduces superclasses), except that the class hierarchy must be acyclic*. So these class declarations are OK:

```
class Functor (m k) => FiniteMap m k where
  ...

class (Monad m, Monad (t m)) => Transform t m where
  lift :: m a -> (t m) a
```

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

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

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

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

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

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

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

5. *The type of each class operation must mention all of the class type variables*. For example:

```
class Coll s a where
```

```
    empty  :: s
    insert :: s -> a -> s
```

is not OK, because the type of `empty` doesn't mention `a`. This rule is a consequence of Rule 1(a), above, for types, and has the same motivation. Sometimes, offending class declarations exhibit misunderstandings. For example, `Coll` might be rewritten

```
  class Coll s a where
    empty  :: s a
    insert :: s a -> a -> s a
```

which makes the connection between the type of a collection of `a`'s (namely `(s a)`) and the element type `a`. Occasionally this really doesn't work, in which case you can split the class like this:

```
  class CollE s where
    empty  :: s

  class CollE s => Coll s a where
    insert :: s -> a -> s
```

7.8.3. 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*.

7.9. Implicit parameters

Implicit paramters are implemented as described in "Implicit parameters: dynamic scoping with static types", J Lewis, MB Shields, E Meijer, J Launchbury, 27th ACM Symposium on Principles of Programming Languages (POPL'00), Boston, Jan 2000.

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

- You can't have an implicit parameter in the context of a class or instance declaration. For example, both these declarations are illegal:

```
class (?x::Int) => C a where ...
instance (?x::a) => Foo [a] where ...
```

Reason: exactly which implicit parameter you pick up depends on exactly where you invoke a function. But the "invocation" of instance declarations is done behind the scenes by the compiler, so it's hard to figure out exactly where it is done. Easiest thing is to outlaw the offending types.

7.10. Functional dependencies

Functional dependencies are implemented as described by Mark Jones in "Type Classes with Functional Dependencies", Mark P. Jones, In Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany, March 2000, Springer-Verlag LNCS 1782.

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

7.11. Explicit 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. 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.

7.11.1. 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.)

7.11.2. 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.)

7.11.3. 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.

7.11.4. 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`.

7.11.5. 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.

7.11.6. 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
```

7.12. Existentially quantified data constructors

The idea of using existential quantification in data type declarations was suggested by Laufer (I believe, thought doubtless someone will correct me), and implemented in Hope+. It's been in Lennart Augustsson's **hbc** Haskell compiler for several years, and proved very useful. Here's the idea. Consider the declaration:

```
data Foo = forall a. MkFoo a (a -> Bool)
```

```
            | Nil
```

The data type `Foo` has two constructors with types:

```
MkFoo :: forall a. a -> (a -> Bool) -> Foo
Nil   :: Foo
```

Notice that the type variable `a` in the type of `MkFoo` does not appear in the data type itself, which is plain `Foo`. For example, the following expression is fine:

```
[MkFoo 3 even, MkFoo 'c' isUpper] :: [Foo]
```

Here, `(MkFoo 3 even)` packages an integer with a function `even` that maps an integer to `Bool`; and `MkFoo 'c' isUpper` packages a character with a compatible function. These two things are each of type `Foo` and can be put in a list.

What can we do with a value of type `Foo`?. In particular, what happens when we pattern-match on `MkFoo`?

```
f (MkFoo val fn) = ???
```

Since all we know about `val` and `fn` is that they are compatible, the only (useful) thing we can do with them is to apply `fn` to `val` to get a boolean. For example:

```
f :: Foo -> Bool
f (MkFoo val fn) = fn val
```

What this allows us to do is to package heterogenous values together with a bunch of functions that manipulate them, and then treat that collection of packages in a uniform manner. You can express quite a bit of object-oriented-like programming this way.

7.12.1. Why existential?

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

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

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

7.12.2. Type classes

An easy extension (implemented in **hbc**) is to allow arbitrary contexts before the constructor. For example:

```
data Baz = forall a. Eq a => Baz1 a a
         | forall b. Show b => Baz2 b (b -> b)
```

The two constructors have the types you'd expect:

```
Baz1 :: forall a. Eq a => a -> a -> Baz
Baz2 :: forall b. Show b => b -> (b -> b) -> Baz
```

But when pattern matching on `Baz1` the matched values can be compared for equality, and when pattern matching on `Baz2` the first matched value can be converted to a string (as well as applying the function to it). So this program is legal:

```
  f :: Baz -> String
  f (Baz1 p q) | p == q    = "Yes"
               | otherwise = "No"
  f (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.

7.12.3. Restrictions

There are several restrictions on the ways in which existentially-quantified constructors can be use.

- When pattern matching, each pattern match introduces a new, distinct, type for each existential type variable. These types cannot be unified with any other type, nor can they escape from the scope of the pattern match. For example, these fragments are incorrect:

```
f1 (MkFoo a f) = a
```

Here, the type bound by `MkFoo` "escapes", because `a` is the result of `f1`. One way to see why this is wrong is to ask what type `f1` has:

```
  f1 :: Foo -> a             - Weird!
```

What is this "`a`" in the result type? Clearly we don't mean this:

```
  f1 :: forall a. Foo -> a   - Wrong!
```

The original program is just plain wrong. Here's another sort of error

```
f2 (Baz1 a b) (Baz1 p q) = a==q
```

It's ok to say `a==b` or `p==q`, but `a==q` is wrong because it equates the two distinct types arising from the two `Baz1` constructors.

- You can't pattern-match on an existentially quantified constructor in a `let` or `where` group of bindings. So this is illegal:

```
f3 x = a==b where { Baz1 a b = x }
```

You can only pattern-match on an existentially-quantified constructor in a `case` expression or in the patterns of a function definition. The reason for this restriction is really an implementation one. Type-checking binding groups is already a nightmare without existentials complicating the picture. Also an existential pattern binding at the top level of a module doesn't make sense, because it's not clear how to prevent the existentially-quantified type "escaping". So for now, there's a simple-to-state restriction. We'll see how annoying it is.

- You can't use existential quantification for `newtype` declarations. So this is illegal:

```
newtype T = forall a. Ord a => MkT a
```

Reason: a value of type `T` must be represented as a pair of a dictionary for `Ord t` and a value of type `t`. That contradicts the idea that `newtype` should have no concrete representation. You can get just the same efficiency and effect by using `data` instead of `newtype`. If there is no overloading involved, then there is more of a case for allowing an existentially-quantified `newtype`, because the `data` because the `data` version does carry an implementation cost, but single-field existentially quantified constructors aren't much use. So the simple restriction (no existential stuff on `newtype`) stands, unless there are convincing reasons to change it.

- You can't use `deriving` to define instances of a data type with existentially quantified data constructors. Reason: in most cases it would not make sense. For example:#

```
data T = forall a. MkT [a] deriving( Eq )
```

To derive `Eq` in the standard way we would need to have equality between the single component of two `MkT` constructors:

```
instance Eq T where
  (MkT a) == (MkT b) = ???
```

But `a` and `b` have distinct types, and so can't be compared. It's just about possible to imagine examples in which the derived instance would make sense, but it seems altogether simpler simply to prohibit such declarations. Define your own instances!

7.13. 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 5.12 in *Haskell Libraries*) for the details.

7.14. Scoped Type Variables

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

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

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

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

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

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

7.14.1. What a pattern type signature means

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

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

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

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

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

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

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

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

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

7.14.2. Scope and implicit quantification

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

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

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

- The type 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).

7.14.3. Result type signatures

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

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

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

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

Result type signatures are not yet implemented in Hugs.

7.14.4. Where a pattern type signature can occur

A pattern type signature can occur in any pattern, but there are restrictions on pattern bindings:

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

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

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

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

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

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

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

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

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

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

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

- Pattern type signatures 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 degenerate 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.

7.15. Pragmas

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

Pragmas all take the form `{-# word ... #-}` where *word* indicates the type of pragma, and is followed optionally by information specific to that type of pragma. Case is ignored in *word*. The various values for *word* that GHC understands are described in the following sections; any pragma encountered with an unrecognised *word* is (silently) ignored.

7.15.1. INLINE pragma

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

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

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

The sledgehammer you can bring to bear is the `INLINE` pragma, used thusly:

```
key_function :: Int -> String -> (Bool, Double)

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

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

The major effect of an `INLINE` pragma is to declare a function's "cost" to be very low. The normal unfolding machinery will then be very keen to inline it.

An `INLINE` pragma for a function can be put anywhere its type signature could be put.

`INLINE` pragmas are a particularly good idea for the `then`/`return` (or `bind`/`unit`) functions in a monad. For example, in GHC's own `UniqueSupply` monad code, we have:

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

7.15.2. NOINLINE pragma

The `NOINLINE` pragma does exactly what you'd expect: it stops the named function from being inlined by the compiler. You shouldn't ever need to do this, unless you're very cautious about code size.

`NOTINLINE` is a synonym for `NOINLINE` (`NOTINLINE` is specified by Haskell 98 as the standard way to disable inlining, so it should be used if you want your code to be portable).

7.15.3. 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, as in:

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

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

Note we use the `RULE` pragma here to indicate that `hammeredLookup` applied at a certain type should be replaced by `blah`. See Section 7.15.6 for more information on `RULES`.

An example in which using `RULES` for specialisation will Win Big:

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

{-# 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.

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

7.15.4. 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.

7.15.5. LINE pragma

This pragma is similar to C's `#line` pragma, and is mainly for use in automatically generated Haskell code. It lets you specify the line number and filename of the original code; for example

```
{-# LINE 42 "Foo.vhs" #-}
```

if you'd generated the current file from something called `Foo.vhs` and this line corresponds to line 42 in the original. GHC will adjust its error messages to refer to the line/file named in the `LINE` pragma.

7.15.6. RULES pragma

The RULES pragma lets you specify rewrite rules. It is described in Section 7.16.

7.15.7. DEPRECATED pragma

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

- You can deprecate an entire module thus:

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

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

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

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

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

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

7.16. Rewrite rules

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

Here is an example:

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

7.16.1. Syntax

From a syntactic point of view:

- Each rule has a name, enclosed in double quotes. The name itself has no significance at all. It is only used when reporting how many times the rule fired.
- There may be zero or more rules in a `RULES` pragma.
- Layout applies in a `RULES` pragma. Currently no new indentation level is set, so you must lay out your rules starting in the same column as the enclosing definitions.
- Each variable mentioned in a rule must either be in scope (e.g. `map`), or bound by the `forall` (e.g. `f`, `g`, `xs`). The variables bound by the `forall` are called the *pattern* variables. They are separated by spaces, just like in a type `forall`.
- A pattern variable may optionally have a type signature. If the type of the pattern variable is polymorphic, it *must* have a type signature. For example, here is the `foldr/build` rule:

  ```
  "fold/build"  forall k z (g::forall b. (a->b->b) -> b -> b) .
  ```

```
foldr k z (build g) = g k z
```

Since `g` has a polymorphic type, it must have a type signature.

- The left hand side of a rule must consist of a top-level variable applied to arbitrary expressions. For example, this is *not* OK:

```
"wrong1"   forall e1 e2.  case True of { True -> e1; False -> e2 } = e1
"wrong2"   forall f.      f True = True
```

In `"wrong1"`, the LHS is not an application; in `"wrong2"`, the LHS has a pattern variable in the head.

- A rule does not need to be in the same module as (any of) the variables it mentions, though of course they need to be in scope.
- Rules are automatically exported from a module, just as instance declarations are.

7.16.2. Semantics

From a semantic point of view:

- Rules are only applied if you use the `-O` flag.
- Rules are regarded as left-to-right rewrite rules. When GHC finds an expression that is a substitution instance of the LHS of a rule, it replaces the expression by the (appropriately-substituted) RHS. By "a substitution instance" we mean that the LHS can be made equal to the expression by substituting for the pattern variables.
- The LHS and RHS of a rule are typechecked, and must have the same type.
- GHC makes absolutely no attempt to verify that the LHS and RHS of a rule have the same meaning. That is undecideable in general, and infeasible in most interesting cases. The responsibility is entirely the programmer's!
- GHC makes no attempt to make sure that the rules are confluent or terminating. For example:

```
"loop"        forall x,y.  f x y = f y x
```

This rule will cause the compiler to go into an infinite loop.

- If more than one rule matches a call, GHC will choose one arbitrarily to apply.
- GHC currently uses a very simple, syntactic, matching algorithm for matching a rule LHS with an expression. It seeks a substitution which makes the LHS and expression syntactically equal modulo alpha conversion. The pattern (rule), but not the expression, is eta-expanded if necessary. (Eta-expanding the epression can lead to laziness bugs.) But not beta conversion (that's called higher-order matching).

  Matching is carried out on GHC's intermediate language, which includes type abstractions and applications. So a rule only matches if the types match too. See Section 7.16.4 below.

- GHC keeps trying to apply the rules as it optimises the program. For example, consider:

```
let s = map f
    t = map g
in
s (t xs)
```

The expression `s (t xs)` does not match the rule `"map/map"`, but GHC will substitute for `s` and `t`, giving an expression which does match. If `s` or `t` was (a) used more than once, and (b) large or a redex, then it would not be substituted, and the rule would not fire.

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

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

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

```
wibble f g xs
```

where `wibble` is defined:

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

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

- All rules are implicitly exported from the module, and are therefore in force in any module that imports the module that defined the rule, directly or indirectly. (That is, if A imports B, which imports C, then C's rules are in force when compiling A.) The situation is very similar to that for instance declarations.

7.16.3. List fusion

The RULES mechanism is used to implement fusion (deforestation) of common list functions. If a "good consumer" consumes an intermediate list constructed by a "good producer", the intermediate list should be eliminated entirely.

The following are good producers:

- List comprehensions
- Enumerations of `Int` and `Char` (e.g. `['a'..'z']`).
- Explicit lists (e.g. `[True, False]`)
- The cons constructor (e.g `3:4:[]`)
- `++`

- `map`
- `filter`
- `iterate`, `repeat`
- `zip`, `zipWith`

The following are good consumers:

- List comprehensions
- `array` (on its second argument)
- `length`
- `++` (on its first argument)
- `foldr`
- `map`
- `filter`
- `concat`
- `unzip`, `unzip2`, `unzip3`, `unzip4`
- `zip`, `zipWith` (but on one argument only; if both are good producers, `zip` will fuse with one but not the other)
- `partition`
- `head`
- `and`, `or`, `any`, `all`
- `sequence_`
- `msum`
- `sortBy`

So, for example, the following should generate no intermediate lists:

```
array (1,10) [(i,i*i) | i <- map (+ 1) [0..9]]
```

This list could readily be extended; if there are Prelude functions that you use a lot which are not included, please tell us.

If you want to write your own good consumers or producers, look at the Prelude definitions of the above functions to see how to do so.

7.16.4. Specialisation

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

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

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

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

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

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

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

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

7.16.5. Controlling what's going on

- Use `-ddump-rules` to see what transformation rules GHC is using.
- Use `-ddump-simpl-stats` to see what rules are being fired. If you add `-dppr-debug` you get a more detailed listing.
- The defintion of (say) `build` in `PrelBase.lhs` looks llike this:

  ```
  build   :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
  {-# INLINE build #-}
  build g = g (:) []
  ```

Notice the `INLINE`! That prevents `(:)` from being inlined when compiling `PrelBase`, so that an importing module will "see" the `(:)`, and can match it on the LHS of a rule. `INLINE` prevents any inlining happening in the RHS of the `INLINE` thing. I regret the delicacy of this.

- In `ghc/lib/std/PrelBase.lhs` look at the rules for `map` to see how to write rules that will do fusion and yet give an efficient program even if fusion doesn't happen. More rules in `PrelList.lhs`.

7.17. Generic classes

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

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

```
  import Generics

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

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

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

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

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

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

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

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

7.17.1. Using generics

To use generics you need to

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

7.17.2. Changes wrt the paper

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

7.17.3. Terminology and restrictions

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

Restrictions:

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

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

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

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

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

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

```
a :+: b
```

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

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

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

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

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

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

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

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

  Here are some example type signatures for generic methods:

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

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

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

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

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

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

7.17.4. Another example

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

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

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

7.18. Concurrent and Parallel Haskell

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

Concurrent and Parallel Haskell have very different purposes.

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

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

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

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

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

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

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

Some details about Parallel Haskell follow. For more information about concurrent Haskell, see Chapter 3 in *Haskell Libraries*.

7.18.1. Features specific to Parallel Haskell

7.18.1.1. The `Parallel` interface (recommended)

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

```
interface Parallel where
infixr 0 `par`
infixr 1 `seq`

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

The expression ``(x `par` y)`` *sparks* the evaluation of `x` (to weak head normal form) and returns `y`. Sparks are queued for execution in FIFO order, but are not executed immediately. At the next heap allocation, the currently executing thread will yield control to the scheduler, and the scheduler will start a new thread (until reaching the active thread limit) for each spark which has not already been evaluated to WHNF.

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

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

```
import Parallel

nfib :: Int -> Int
nfib n | n <= 1 = 1
       | otherwise = par n1 (seq n2 (n1 + n2 + 1))
                     where n1 = nfib (n-1)
                           n2 = nfib (n-2)
```

For values of `n` greater than 1, we use `par` to spark a thread to evaluate `nfib (n-1)`, and then we use `seq` to force the parent thread to evaluate `nfib (n-2)` before going on to add together these two subexpressions. In this divide-and-conquer approach, we only spark a new thread for one branch of the computation (leaving the parent to evaluate the other branch). Also, we must use `seq` to ensure that the parent will evaluate `n2` *before* `n1` in the expression `(n1 + n2 + 1)`. It is not sufficient to reorder the expression as `(n2 + n1 + 1)`, because the compiler may not generate code to evaluate the addends from left to right.

7.18.1.2. Underlying functions and primitives

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

7.18.1.3. Scheduling policy for concurrent threads

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

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

7.18.1.4. Scheduling policy for parallel threads

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

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

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

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

7.19.1. Divergence from Haskell 98

7.19.1.1. Lexical syntax

- The Haskell report specifies that programs may be written using Unicode. GHC only accepts the ISO-8859-1 character set at the moment.
- Certain lexical rules regarding qualified identifiers are slightly different in GHC compared to the Haskell report. When you have `module.reservedop`, such as `M.\`, GHC will interpret it as a single qualified operator rather than the two lexemes `M` and `.\`.
- When `-fglasgow-exts` is on, GHC reserves several keywords beginning with two underscores. This is due to the fact that GHC uses the same lexical analyser for interface file parsing as it does for source file parsing, and these keywords are used in interface files. Do not use any identifiers beginning with a double underscore in `-fglasgow-exts` mode.

7.19.1.2. Context-free syntax

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

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

and parses as:

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

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

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

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

7.19.1.3. Expressions and patterns

Very long `String` constants:

May not go through. If you add a "string gap" every few thousand characters, then the strings can be as long as you like.

Bear in mind that string gaps and the `-cpp` option don't mix very well (see Section 4.12.3).

7.19.1.4. Declarations and bindings

None known.

7.19.1.5. Module system and interface files

Namespace pollution

Several modules internal to GHC are visible in the standard namespace. All of these modules begin with `Prel`, so the rule is: don't use any modules beginning with `Prel` in your program, or you may be comprehensively screwed.

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

Multiply-defined array elements—not checked:

This code fragment *should* elicit a fatal error, but it does not:

```
main = print (array (1,1) [(1,2), (1,3)])
```

7.19.1.7. In Prelude support

The `Char` type

The Haskell report says that the `Char` type holds 16 bits. GHC follows the ISO-10646 standard a little more closely: `maxBound :: Char` in GHC is `0x10FFFF`.

Arbitrary-sized tuples:

Tuples are currently limited to size 61. HOWEVER: standard instances for tuples (`Eq`, `Ord`, `Bounded`, `Ix Read`, and `Show`) are available *only* up to 5-tuples.

This limitation is easily subvertible, so please ask if you get stuck on it.

7.19.2. GHC's interpretation of undefined behaviour in Haskell 98

This section documents GHC's take on various issues that are left undefined or implementation specific in Haskell 98.

Sized integral types

In GHC the `Int` type follows the size of an address on the host architecture; in other words it holds 32 bits on a 32-bit machine, and 64-bits on a 64-bit machine.

Arithmetic on `Int` is unchecked for overflow, so all operations on `Int` happen modulo 2^n where n is the size in bits of the `Int` type.

The `fromInteger`function (and hence also `fromIntegral`) is a special case when converting to `Int`. The value of `fromIntegral x :: Int` is given by taking the lower n bits of `(abs x)`, multiplied by the sign of `x` (in 2's complement n-bit arithmetic). This behaviour was chosen so that for example writing `0xffffffff :: Int` preserves the bit-pattern in the resulting `Int`.

Unchecked float arithmetic

Operations on `Float` and `Double` numbers are *unchecked* for overflow, underflow, and other sad occurrences. (note, however that some architectures trap floating-point overflow and loss-of-precision and report a floating-point exception, probably terminating the program).

Chapter 8. Foreign function interface

The foreign function interface consists of the following components:

- The Foreign Function Interface language specification (which constitutes most of this Chapter, beginning with Section 8.1). You must use the `-fglasgow-exts` command-line option to make GHC understand the `foreign` declarations defined by the FFI.
- Several library modules which provide access to types used by foreign languages and utilties for marshalling values to and from foreign functions, and for converting errors in the foreign language into Haskell IO errors. See Section 5.13 in *Haskell Libraries* for more details.

8.1. 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.

In the following, we will discuss the language extensions of the FFI. The extensions 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.

8.2. 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 8.2.5 specifies the mapping from Haskell types to external types.

8.2.1. 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 | ' )*
```

8.2.2. Naming the external function

The name of the external function is a string:

```
ext_fun  : string
```

For example,

```
foreign import stdcall "RegCloseKey" regCloseKey :: Ptr a -> IO ()
```

states that the external function named `RegCloseKey` should be bound to the Haskell name `regCloseKey`.

The details of where exactly the external name can be found, such as whether or not it is dynamically linked, and which library it might come from, are implementation dependent. This information is expected to be provided using a compiler-specific method (eg. GHC uses either packages or command-line options to specify libraries and extra include files).

If the Haskell name of the imported function is identical to the external name, the `ext_fun` can be omitted. e.g.:

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

is identical to

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

8.2.3. 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.

8.2.4. 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.

Section 8.2.4.2 defines `prim_result`; Section 8.2.4.1 defines `prim_arg`.

8.2.4.1. 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 | ForeignPtr a

new_ty : a Haskell newtype of a prim_arg.

ext_ty : int_ty   | word_ty | float_ty
       | Ptr a    | 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 8.2.5 for details.
- `prim_arg` represent the set of permissible argument types. In addition to `ext_ty`, `ForeignPtr` is also included. The `ForeignPtr` type represent values that are pointers to some external entity/object. It differs from the `Ptr` type in that `ForeignPtr`s are *finalized*, i.e., once the garbage collector determines that a `ForeignPtr` is unreachable, it will invoke a finalising procedure attached to the `ForeignPtr` 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.
- A 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.)

8.2.4.2. 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 `ForeignPtr`s is not supported/allowed.
- Haskell newtypes that wrap up `ext_ty` are also permitted.

8.2.5. 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 8-1 defines the mapping between Haskell and C types.

Table 8-1. 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)
`Ptr a`	`HsPtr`	void* (6)	
`ForeignPtr a`	`HsForeignPtr`	void* (7)	
`StablePtr a`	`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 `Ptr` by the FFI, so there is again the guarantee that

HsForeignPtr 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*.

8.2.6. 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 8.4 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 8.2 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 imported 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 :: ForeignPtr a -> IO ()
```

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

The `ForeignPtr` 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.

8.3. 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.

8.4. 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 export`ed IO action must catch all exceptions, as the FFI does not address how to signal Haskell exceptions to the outside world.

8.4.1. 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 8.2.5.

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 ()
```

8.4.2. 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 label`s:

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

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

8.5. Using the FFI with GHC

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

8.5.1. Using `foreign export` with GHC

When GHC compiles a module (say `M.hs`) which uses `foreign export` or `foreign export dynamic`, it generates two additional files, `M_stub.c` and `M_stub.h`. GHC will automatically compile `M_stub.c` to generate `M_stub.o` at the same time.

For a plain `foreign export`, the file `M_stub.h` contains a C prototype for the foreign exported function, and `M_stub.c` contains its definition. For example, if we compile the following module:

```
module Foo where

foreign export foo :: Int -> IO Int

foo :: Int -> IO Int
foo n = return (length (f n))
```

```
f :: Int -> [Int]
f 0 = []
f n = n:(f (n-1))
```

Then `Foo_stub.h` will contain something like this:

```
#include "HsFFI.h"
extern HsInt foo(HsInt a0);
```

and `Foo_stub.c` contains the compiler-generated definition of `foo()`. To invoke `foo()` from C, just `#include "Foo_stub.h"` and call `foo()`.

8.5.1.1. Using your own `main()`

Normally, GHC's runtime system provides a `main()`, which arranges to invoke `Main.main` in the Haskell program. However, you might want to link some Haskell code into a program which has a main function written in another languagem, say C. In order to do this, you have to initialize the Haskell runtime system explicitly.

Let's take the example from above, and invoke it from a standalone C program. Here's the C code:

```
#include <stdio.h>
#include "foo_stub.h"

#include "RtsAPI.h"

extern void __stginit_Foo ( void );

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

  startupHaskell(argc, argv, __stginit_Foo);

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

  shutdownHaskell();

  return 0;
}
```

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

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

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

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

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

NOTE: when linking the final program, it is normally easiest to do the link using GHC, although this isn't essential. If you do use GHC, then don't forget the flag `-no-hs-main`, otherwise GHC will try to link to the `Main` Haskell module.

8.5.1.2. Using `foreign export dynamic` with GHC

When `foreign export dynamic` is used in a Haskell module, The C stub file `M_stub.c` generated by GHC contains small helper functions used by the code generated for the `foreign export dynamic`, so it must be linked in to the final program. When linking the program, remember to include `M_stub.o` in the final link command line, or you'll get link errors for the missing function(s) (this isn't necessary when building your program with `ghc --make`, as GHC will automatically link in the correct bits).

8.5.2. Using function headers

When generating C (using the `-fvia-C` directive), one can assist the C compiler in detecting type errors by using the `-#include` directive (Section 4.12.4) 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 8-1.

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.

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's 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 9. What to do when something goes wrong

If you still have a problem after consulting this section, then you may have found a *bug*—please report it! See Section 1.2 for details on how to report a bug and a list of things we'd like to know about your bug. If in doubt, send a report—we love mail from irate users :-!

(Section 7.19, which describes Glasgow Haskell's shortcomings vs. the Haskell language definition, may also be of interest.)

9.1. When the compiler "does the wrong thing"

"Help! The compiler crashed (or 'panic'd)!"

These events are *always* bugs in the GHC system—please report them.

"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.

"Is this line number right?"

On this score, GHC usually does pretty well, especially if you "allow" it to be off by one or two. In the case of an instance or class declaration, the line number may only point you to the declaration, not to a specific method.

Please report line-number errors that you find particularly unhelpful.

9.2. When your program "does the wrong thing"

(For advice about overly slow or memory-hungry Haskell programs, please see Chapter 6).

"Help! My program crashed!"

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

If your program has no foreign calls in it, then a crash is always a BUG in the GHC system, except in one case: If your program is made of several modules, each module must have been compiled after any modules on which it depends (unless you use `.hi-boot` files, in which case these *must* be correct with respect to the module source).

For example, if an interface is lying about the type of an imported value then GHC may well generate duff code for the importing module. *This applies to pragmas inside interfaces too!* If the pragma is lying (e.g., about the "arity" of a value), then duff code may result. Furthermore, arities may change even if types do not.

In short, if you compile a module and its interface changes, then all the modules that import that interface *must* be re-compiled.

A useful option to alert you when interfaces change is `-hi-diffs`. It will run **diff** on the changed interface file, before and after, when applicable.

If you are using **make**, GHC can automatically generate the dependencies required in order to make sure that every module *is* up-to-date with respect to its imported interfaces. Please see Section 4.9.5.1.

If you are down to your last-compile-before-a-bug-report, we would recommend that you add a `-dcore-lint` option (for extra checking) to your compilation options.

So, before you report a bug because of a core dump, you should probably:

```
% rm *.o        # scrub your object files
% make my_prog  # re-make your program; use -hi-
diffs to highlight changes;
                # as mentioned above, use -dcore-
lint to be more paranoid
% ./my_prog ... # retry...
```

Of course, if you have foreign calls in your program then all bets are off, because you can trash the heap, the stack, or whatever.

If you are interested in hard-core debugging of a crashing GHC-compiled program, please see Section 9.3.

"My program entered an 'absent' argument."

This is definitely caused by a bug in GHC. Please report it (see Section 1.2).

"What's with this 'arithmetic (or 'floating') exception' "?

`Int`, `Float`, and `Double` arithmetic is *unchecked*. Overflows, underflows and loss of precision are either silent or reported as an exception by the operating system (depending on the architecture). Divide-by-zero *may* cause an untrapped exception (please report it if it does).

9.3. Hard-core debugging of GHC-compiled programs

If your program is crashing, you should almost surely file a bug report, as outlined in Section 1.2.

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 10. Other Haskell utility programs

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

10.1. "Yacc for Haskell": happy

Andy Gill and Simon Marlow have written a parser-generator for Haskell, called **happy**. **Happy** is to Haskell what **Yacc** is to C.

You can get **happy** from the Happy Homepage (http://www.haskell.org/happy/).

Happy is at its shining best when compiled by GHC.

10.2. Writing Haskell interfaces to C code: hsc2hs

The **hsc2hs** command can be used to automate some parts of the process of writing Haskell bindings to C code. It reads an almost-Haskell source with embedded special constructs, and outputs a real Haskell file with these constructs processed, based on information taken from some C headers. The extra constructs deal with accessing C data from Haskell.

It may also output a C file which contains additional C functions to be linked into the program, together with a C header that gets included into the C code to which the Haskell module will be compiled (when compiled via C) and into the C file. These two files are created when the `#def` construct is used (see below).

Actually **hsc2hs** does not output the Haskell file directly. It creates a C program that includes the headers, gets automatically compiled and run. That program outputs the Haskell code.

In the following, "Haskell file" is the main output (usually a `.hs` file), "compiled Haskell file" is the Haskell file after **ghc** has compiled it to C (i.e. a `.hc` file), "C program" is the program that outputs the Haskell file, "C file" is the optionally generated C file, and "C header" is its header file.

10.2.1. Command line syntax

hsc2hs takes input files as arguments, and flags that modify its behavior:

`-t FILE` or `--template=FILE`

The template file (see below).

`-c PROG` or `--cc=PROG`

The C compiler to use (default: **ghc**)

`-l PROG` or `--ld=PROG`

The linker to use (default: **gcc**).

`-C FLAG` or `--cflag=FLAG`

An extra flag to pass to the C compiler.

`-I DIR`

Passed to the C compiler.

`-L FLAG` or `--lflag=FLAG`

An extra flag to pass to the linker.

`-i FILE` or `--include=FILE`

As if the appropriate `#include` directive was placed in the source.

`-D NAME[=VALUE]` or `--define=NAME[=VALUE]`

As if the appropriate `#define` directive was placed in the source.

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

Name of the Haskell file.

`--help`

Display a summary of the available flags.

`--version`

Output version information.

`--no-compile`

Stop after writing out the intermediate C program to disk. The file name for the intermediate C program is the input file name with `.hsc` replaced with `_hsc_make.c`.

The input file should end with .hsc. Output files by default get names with the `.hsc` suffix replaced:

`.hs`	Haskell file
`_hsc.h`	C header
`_hsc.c`	C file

The C program is compiled using the Haskell compiler. This provides the include path to `HsFFI.h` which is automatically included into the C program.

10.2.2. Input syntax

All special processing is triggered by the `#` operator. To output a literal `#`, write it twice: `##`. Inside string literals and comments `#` characters are not processed.

A `#` is followed by optional spaces and tabs, an alphanumeric keyword that describes the kind of processing, and its arguments. Arguments look like C expressions separated by commas (they are not written inside parens). They extend up to the nearest unmatched `)`, `]` or `}`, or to the end of line if it occurs outside any `() [] {} '' "" /**/` and is not preceded by a backslash. Backslash-newline pairs are stripped.

In addition `#{stuff}` is equivalent to `#stuff` except that it's self-delimited and thus needs not to be placed at the end of line or in some brackets.

Meanings of specific keywords:

`#include <file.h>`
`#include "file.h"`

The specified file gets included into the C program, the compiled Haskell file, and the C header. `<HsFFI.h>` is included automatically.

`#define name`
`#define name value`
`#undef name`

Similar to `#include`. Note that `#includes` and `#defines` may be put in the same file twice so they should not assume otherwise.

`#let name parameters = "definition"`

Defines a macro to be applied to the Haskell source. Parameter names are comma-separated, not inside parens. Such macro is invoked as other `#`-constructs, starting with `#name`. The definition will be put in the C program inside parens as arguments of `printf`. To refer to a parameter, close the quote, put a parameter name and open the quote again, to let C string literals concatenate. Or use `printf`'s format directives. Values of arguments must be given as strings, unless the macro stringifies them itself using the C preprocessor's `#parameter` syntax.

`#def C_definition`

The definition (of a function, variable, struct or typedef) is written to the C file, and its prototype or extern declaration to the C header. Inline functions are handled correctly. struct definitions and typedefs are written to the C program too. The `inline`, `struct` or `typedef` keyword must come just after `def`.

`#if condition`
`#ifdef name`
`#ifndef name`
`#elif condition`
`#else`

`#endif`
`#error message`
`#warning message`

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

`#const C_expression`

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

`#const_str C_expression`

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

`#type C_type`

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

`#peek struct_type, field`

A function that peeks a field of a C struct will be output. It will have the type `Storable b => Ptr a -> IO b`. The intention is that `#peek` and `#poke` can be used for implementing the operations of class `Storable` for a given C struct (see Section 5.35 in *Haskell Libraries*).

`#poke struct_type, field`

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

`#ptr struct_type, field`

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

`#enum type, constructor, value, value, ...`

A shortcut for multiple definitions which use `#const`. Each `value` is a name of a C integer constant, e.g. enumeration value. The name will be translated to Haskell by making each letter following an underscore uppercase, making all the rest lowercase, and removing underscores. You can supply a different translation by writing `hs_name = c_value` instead of a `value`, in which case `c_value` may be an arbitrary expression. The `hs_name` will be defined as having the specified `type`. Its definition is the specified `constructor` (which in fact may be an expression or be empty) applied to the appropriate integer value. You can have multiple `#enum` definitions with the same `type`; this construct does not emit the type definition itself.

10.2.3. Custom constructs

`#const`, `#type`, `#peek`, `#poke` and `#ptr` are not hardwired into the **hsc2hs**, but are defined in a C template that is included in the C program: `template-hsc.h`. Custom constructs and templates can be used too. Any `#`-construct with unknown key is expected to be handled by a C template.

A C template should define a macro or function with name prefixed by `hsc_` that handles the construct by emitting the expansion to stdout. See `template-hsc.h` for examples.

Such macros can also be defined directly in the source. They are useful for making a `#let`-like macro whose expansion uses other `#let` macros. Plain `#let` prepends `hsc_` to the macro name and wraps the defininition in a `printf` call.

Chapter 11. Building and using Win32 DLLs

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

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

11.1. Linking with DLLs

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

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

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

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

11.2. Not linking with DLLs

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

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

11.3. Creating a DLL

Making libraries into DLLs doesn't work on Windows at the moment (and is no longer supported); however, all the machinery is still there. If you're interested, contact the GHC team. Note that building an entire Haskell application as a DLL is still supported (it's just inter-DLL Haskell calls that don't work). Sealing up your Haskell library inside a DLL is straightforward; compile up the object files that make up the library, and then build the DLL by issuing a command of the form:

```
ghc --mk-dll -o foo.dll bar.o baz.o wibble.a -lfooble
```

By feeding the ghc compiler driver the option `--mk-dll`, it will build a DLL rather than produce an executable. The DLL will consist of all the object files and archives given on the command line.

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

A couple of things to notice:

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

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

See Microsoft documentation for details, but a module definition file simply lists what entry points you want to export. Here's one that's suitable when building a Haskell COM server DLL:

```
EXPORTS
 DllCanUnloadNow      = DllCanUnloadNow@0
 DllGetClassObject    = DllGetClassObject@12
 DllRegisterServer    = DllRegisterServer@0
 DllUnregisterServer = DllUnregisterServer@0
```

- In addition to creating a DLL, the `--mk-dll` option also creates an import library. The import library name is derived from the name of the DLL, as follows:

  ```
  DLL: HScool.dll  ==> import lib: libHScool_imp.a
  ```

The naming scheme may look a bit weird, but it has the purpose of allowing the co-existence of import libraries with ordinary static libraries (e.g., `libHSfoo.a` and `libHSfoo_imp.a`. Additionally, when the compiler driver is linking in non-static mode, it will rewrite occurrence of `-lHSfoo` on the command line to `-lHSfoo_imp`. By doing this for you, switching from non-static to static linking is simply a question of adding `-static` to your command line.

11.4. Making DLLs to be called from other languages

If you want to package up Haskell code to be called from other languages, such as Visual Basic or C++, there are some extra things it is useful to know. The dirty details are in the *Foreign Function Interface* definition, but it can be tricky to work out how to combine this with DLL building, so here's an example:

- Use `foreign export` declarations to export the Haskell functions you want to call from the outside. For example,

  ```
  module Adder where

  adder :: Int -> Int -> IO Int  - gratuitous use of IO
  adder x y = return (x+y)

  foreign export stdcall adder :: Int -> Int -> IO Int
  ```

- Compile it up:

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

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

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

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

  EXTFUN(__stginit_Adder);

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

Here, `Adder` is the name of the root module in the module tree (as mentioned above, there must be a single root module, and hence a single module tree in the DLL). Compile this up:

```
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.

To make both writing new libraries and reading programs using them easier, a few conventions the set of interfaces specified in this document try to adhere to are listed in the following sections. Note that because of compatibility issues and the evolving nature of this library collection, there are still a few exceptions, but hopefully they vanish in the future.

These conventions are not carved into stone for eternity, they are just a collection of thoughts people involved in the development of these libraries had at some point in time. They were formed after quite some discussions and are heavily based on an initial suggestion of Marcin Kowalczyk <`qrczak@knm.org.pl`>. Feel free to discuss them, the Haskell mailing list <`haskell@haskell.org`> is probably the right place for this.

1.1. Usage

If you're using hslibs with GHC[i], then you need to add `-package p` to the command line for each package from which you're using a module. See Section 4.10 in *The Glasgow Haskell Compiler User's Guide, Version 5.02* for an explanation of packages.

1.2. Naming conventions

This is what people like discussing about most: names. Note that the conventions are not mutually exclusive, e.g. should the function creating a set from a list of elements have the name `set` or `listToSet`? (Alas, it currently has neither name.)

The following nomenclature is used: Pure, i.e. non-monadic functions are simply called, well, *functions*. Monadic functions, i.e. functions having a type `... -> m a` for some Monad `m` are called *actions*.

1.2.1. Constructor names

- Empty values of type `X` have the name `emptyX`, e.g. `emptySet`.
- Actions creating a new empty value of type `X` have the name `newEmptyX`, e.g. `newEmptyMVar`.
- Functions creating an arbitrary value of type `X` have the name `X` itself, e.g. `array`. (*TODO*: This often collides with `xToY` convention, how should this be resolved?)

- Actions creating new values arbitrary values of type `X` have the name `newX`, e.g. `newIORef`.

1.2.2. Accessor names

- Functions getting an attribute of an value or a part of it have the name of the attribute itself, e.g. `length`, `bounds`.
- Actions accessing some kind of reference or state have the name `getX`, where `X` is the type of the contents or the name of the part being accessed, e.g. `getChar`, `getEnv`. An alternative naming scheme is `readY`, where `Y` is the type of the reference or container, e.g. `readIORef`.
- Functions or actions getting a value via a pointer-like type `X` should be named `deRefX`, e.g. `deRefStablePtr`, `deRefWeak`.

1.2.3. Modificator names

- Functions returning an value with attribute `X` set to a new value should be named `setX`. (*TODO*: Add Examples.)
- Actions setting some kind of reference or state have the name `putX`, where `X` is the type of the contents or the name of the part being accessed, e.g. `putChar`. An alternative naming scheme is `writeY`, where `Y` is the type of the reference or container, e.g. `writeIORef`.
- Actions in the `IO` monad setting some global state `X` are traditionally named `setX`, too, although `putX` would be more appropriate, e.g. `y`, `setReadlineName`.
- Functions returning a container `X` modified by a function of type `a -> a` have the name `updateX`. (*TODO*: Add Examples.)
- Actions modifying a container `X` by a function of type `a -> a` have the name `modifyX`, e.g. `modifySTRef`.

1.2.4. Predicate names

- Predicates, both non-monadic and monadic, testing a property `X` have the name `isX`.

1.2.5. Names for conversions

- Functions converting a value of type `X` to a value of type `Y` have the name `xToY` with all leading uppercase characters of `X` converted to lower case, e.g. `stToIO`.
- Overloaded conversion functions of type `C a => a -> X` have the name `toX`, e.g. `toInteger`.
- Overloaded conversion functions of type `C a => X -> a` have the name `fromX`, e.g. `fromInteger`.

1.2.6. Miscellaneous naming conventions

- An action that is identical to another one called `X`, but discards the return value has the name `X_`, e.g. `mapM` and `mapM_`.
- Functions and actions which are potentially dangerous to use and leave some kind of proof obligation to the programmer have the name `unsafeX`, e.g. `unsafePerformIO`.
- There are two conventions for binary and N-ary variants of an associative operation: One convention uses an operator or a short name for the binary operation and a long name for the N-ary variant, e.g. `(+)` and `sum`, `max` and `maximum`. The other convention suffixes the N-ary variant with `Many`. (*TODO*: Add Examples.)
- If possible, names are chosen such that either plain application or `` arg1 `operation` arg2 `` is correct English, e.g. `isPrefixOf` is good for use in backquotes.

1.3. Miscellaneous conventions

- Actions setting and modifying a kind of reference or state return `()`, getting the value is separate, e.g. `writeIORef` and `modifyIORef` both return `()`, only `readIORef` returns the value in an `IORef`
- A function or action taking a some kind of state and returning a pair consisting of a result and a new state, the result is the first element of the pair and the new state is the second, see e.g. `Random`
- A module defining a data type `X` has the itself the name `X`, e.g. `StablePtr`. For convenience there are modules simply re-exporting modules which are often used together, e.g. `Foreign`.
- When the type `Either` is used to encode an error condition and a normal result, `Left` is used for the former and `Right` for the latter, see e.g. `MonadEither`.

- A module corresponding to a class (e.g. `Bits`) contains the class definition, perhaps some auxiliary functions, and all sensible instances for Prelude types, but nothing more. Other modules containing types for which an instance for the class in question makes sense contain the code for the instance itself.
- Record-like C bit fields or structs have a record-like interface, i.e. pure getting and setting of fields. (*TODO*: Clarify a little bit. Add examples.)
- Although the possibility of partial application suggests the type `attr -> object -> object` for functions setting an attribute or value, infix notation with backquotes implies `object -> attr -> object`. (*TODO*: Add Examples.)

1.4. Known inconsistencies

TODO.

Chapter 2. The Prelude and the standard (std) package

The prelude is documented in the Haskell 98 report (http://www.haskell.org/onlinereport/) and the standard libraries in the Haskell 98 libraries report (http://www.haskell.org/onlinelibrary/).
Low-level deviations of GHC from Haskell '98 are noted in Section 7.19 in *The Glasgow Haskell Compiler User's Guide, Version 5.02*; deviations from the standard libraries are noted below.

2.1. Bugs, differences and caveats in GHC's standard library implementation

2.1.1. General

File I/O on Windows

On Windows, reading a file in text mode (which is the default) will translate CRLF to LF, and writing will translate LF to CRLF. This is usually what you want with text files. With binary files this is undesirable; also, as usual under Microsoft operating systems, text mode treats control-Z as EOF.

2.1.2. `System`

`system` on Windows

is implemented using Windows's native `system` call, which ignores the `SHELL` environment variable, and always passes the command to the Windows command interpreter (`cmd.exe` or `command.com`), hence Unixy shell tricks will not work.

`progName`

The report says that this function should return the name of the program "as invoked", which might suggest to Unix types that it should return the equivalent of `argv[0]`. However, this is hard-to-impossible on some non-Unix OSes, so instead, for maximum portability, we just return the leafname of the program as invoked. Even then there are some differences between platforms: on Windows, for example, a program invoked as `foo` is probably really `foo.exe`, and that is what `progName` will return.

Chapter 3. The `concurrent` package: concurrency support

3.1. 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 `MVar`s, a kind of synchronised mutable variable (see Section 3.4.3). Several common concurrency abstractions can be built from `MVar`s, and these are provided by the `Concurrent` library, which is described in the later sections. Threads may also communicate via exceptions.

3.2. 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 `ThreadId`s.

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 5.12) may be raised in any thread for which we have a `ThreadId`, with `throwTo`:

```
throwTo :: 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 `throwTo` (and therefore `killThread`) is that they are *synchronous*, in the sense that after performing a `throwTo` 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.

3.3. Scheduling

Scheduling may be either pre-emptive or co-operative, depending on the implementation of Concurrent Haskell (see below for imformation related to specific compilers). In a co-operative system, 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 `as` and `bs`. 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 ()
```

3.3.1. 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.

3.3.2. 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.

3.4. Concurrency abstractions

3.4.1. `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 ()
```

3.4.2. `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
```

3.4.3. `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)
tryPutMVar       :: MVar a -> a -> IO Bool
isEmptyMVar      :: MVar a -> IO Bool
withMVar  :: MVar a -> (a -> IO b) -> IO b
modifyMVar_   :: MVar a -> (a -> IO a) -> IO ()
modifyMVar   :: MVar a -> (a -> IO (a,b)) -> IO b
addMVarFinalizer :: MVar a -> IO () -> IO ()
```

`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` will cause the calling thread to block until the `MVar` becomes empty again (NOTE: this is a change from the previous behaviour, where `putMVar` on a full `MVar` would raise an exception).

`tryTakeMVar`

The `tryTakeMVar` function 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`.

`tryPutMVar`

The `tryPutMVar` function is a non-blocking version of `putMVar`. If the `MVar` is empty, then it behaves as `putMVar`, and returns `True`. If the `MVar` is full, instead of blocking it returns `False` immediately.

`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.

`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).

`withMVar`

`withMVar` is a safe wrapper for operating on the contents of an `MVar`. This operation is exception-safe: it will replace the `MVar` if an exception is raised (see Section 5.12).

`modifyMVar`
`modifyMVar_`

`modifyMVar` and `modifyMVar_` are safe wrappers for replacing the contents of an `MVar` with a new value. These operations are exception-safe: in the event that an exception is raised during the operation, the old contents of the `MVar` will be replaced.

`addMVarFinalizer`

`addMVarFinalizer` places a finalizer on an `MVar` object, in the same way as `addForeignPtrFinalizer` (see Section 5.15).

3.4.4. `QSem`: General semaphores

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

3.4.5. `QSemN`: Quantity semaphores

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

3.4.6. `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 ()
```

3.4.7. 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.

3.5. 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
```

3.6. GHC-specific concurrency issues

This section describes features specific to GHC's implementation of Concurrent Haskell.

3.6.1. Terminating the program

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 $
  ...
```

The main thread principle also applies to calls to Haskell from outside, using `foreign export`. When the `foreign export`ed function is invoked, it starts a new main thread, and it returns when this main thread terminates. If the call causes new threads to be forked, they may remain in the system after the `foreign export`ed function has returned.

3.6.2. Pre-emption

GHC implements pre-emptive multitasking: the execution of threads are interleaved in a random fashion. More specifically, a thread may be pre-empted whenever it allocates some memory, which unfortunately means that tight loops which do no allocation tend to lock out other threads (this only seems to happen with pathalogical benchmark-style code, however).

The rescheduling timer runs on a 20ms granularity by default, but this may be altered using the `-i<n>` RTS option. After a rescheduling "tick" the running thread is pre-empted as soon as possible.

One final note: the `aaaa`/`bbbb` example may not work too well on GHC (see Section 3.3), due to the locking on a `Handle`. Only one thread may hold the lock on a `Handle` at any one time, so if a reschedule happens while a thread is holding the lock, the other thread won't be able to run. The upshot is that the switch from `aaaa` to `bbbbb` happens infrequently. It can be improved by lowering the reschedule tick period. We also have a patch that causes a reschedule whenever a thread waiting on a lock is woken up, but haven't found it to be useful for anything other than this example :-)

Chapter 4. The `data` package: datatypes

4.1. Edison

Edison is a complete package of data structures for Haskell. Documentation is available online (http://www.haskell.org/ghc/docs/edison/).

4.2. 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 => (elt1 -> elt2 -> elt3)
                           -> FiniteMap key elt1 -> FiniteMap key elt2 -
> FiniteMap key elt3

-      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]
```

4.3. `Set`

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

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
addToSet       :: Ord a => Set a -> a -> Set a
delFromSet     :: Ord a => Set a -> a -> Set a

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

cardinality     :: Set a -> Int
```

Chapter 5. The `lang` package: language support

5.1. `Addr`

This library is DEPRECATED. Use `Ptr` (see Section 5.29) instead.

5.1.1. Address Type and Arithmetic

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

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

The following specifies the behaviour of the four function definitions.

`nullAddr :: Addr`

The constant `nullAddr` contains a distinguished value of `Addr` that is not 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 -> Int -> Addr`

Advances the given address by the given address offset.

`minusAddr :: Addr -> Addr -> Int`

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

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

5.1.2. 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 */
```

5.2. Bits

This module defines bitwise operations for signed and unsigned ints. Instances of `class Bits` for the `Int` and `Integer` types are available from this module, and instances for explicitly sized integral types are available from the `Int` (Section 5.18) and `Word` (Section 5.39) modules.

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

class Num a => 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

  bit i             = shift 0x1 i
  setBit x i        = x .|. bit i
  clearBit x i      = x .&. complement (bit i)
  complementBit x i = x `xor` bit i
  testBit x i       = (x .&. bit i) /= 0

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)

instance Bits Int
instance Bits Integer
```

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.
- `rotate` is well defined only if `bitSize` is also well defined (`bitSize` is undefined for `Integer`, for example).
- `bit i` is the value with the i'th bit set.

5.3. ByteArray

NOTE: The `ByteArray` interface is deprecated, please use `IArray` (Section 5.17) or `MArray` (Section 5.21) 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.

5.4. 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.

5.5. CError

The module CError facillitates C-specific error handling, which essentially entails handling of the values of the variable errno. All definitions are together with the remainder of the C-specific marshalling support also available via the module CForeign (Section 5.6).

In Haskell land, values of the C errno are represented by the type Errno.

```
newtype Errno = Errno CInt  - instances: Eq
```

The implementation of Errno is disclosed on purpose. Different operating systems and/or C libraries often support different values of errno. This module defines the common values, but due to the open definition of Errno users may add definition, which are not predefined. The predefined values are the following:

```
eOK, e2BIG, eACCES, eADDRINUSE, eADDRNOTAVAIL, eADV, eAFNOSUPPORT, eAGAIN,
  eALREADY, eBADF, eBADMSG, eBADRPC, eBUSY, eCHILD, eCOMM, eCONNABORTED,
  eCONNREFUSED, eCONNRESET, eDEADLK, eDESTADDRREQ, eDIRTY, eDOM, eDQUOT,
  eEXIST, eFAULT, eFBIG, eFTYPE, eHOSTDOWN, eHOSTUNREACH, eIDRM, eILSEQ,
  eINPROGRESS, eINTR, eINVAL, eIO, eISCONN, eISDIR, eLOOP, eMFILE, eMLINK,
  eMSGSIZE, eMULTIHOP, eNAMETOOLONG, eNETDOWN, eNETRESET, eNETUNREACH,
  eNFILE, eNOBUFS, eNODATA, eNODEV, eNOENT, eNOEXEC, eNOLCK, eNOLINK,
  eNOMEM, eNOMSG, eNONET, eNOPROTOOPT, eNOSPC, eNOSR, eNOSTR, eNOSYS,
  eNOTBLK, eNOTCONN, eNOTDIR, eNOTEMPTY, eNOTSOCK, eNOTTY, eNXIO,
  eOPNOTSUPP, ePERM, ePFNOSUPPORT, ePIPE, ePROCLIM, ePROCUNAVAIL,
  ePROGMISMATCH, ePROGUNAVAIL, ePROTO, ePROTONOSUPPORT, ePROTOTYPE,
  eRANGE, eREMCHG, eREMOTE, eROFS, eRPCMISMATCH, eRREMOTE, eSHUTDOWN,
  eSOCKTNOSUPPORT, eSPIPE, eSRCH, eSRMNT, eSTALE, eTIME, eTIMEDOUT,
  eTOOMANYREFS, eTXTBSY, eUSERS, eWOULDBLOCK, eXDEV
  :: Errno
```

The meaning of these values corresponds to that of the C constants of the same name - after dropping the leading "e".

The Eq instance of Errno is also OS-dependent as it is only defined for valid values of Errno. The validity of these values is determined by the following function:

```
isValidErrno :: Errno -> Bool
```

The functions `isValidErrno` yields `True` for all values of `errno` that have a defined meaning on the present system.

```
getErrno   :: IO Errno
resetErrno :: IO ()
```

The two functions `getErrno` and `resetErrno` obtain the current value of the `errno` variable and reset it to `eOK`, respectively. On Haskell system with thread support, these and the following operations manipulate thread-local instances of `errno`.

```
errnoToIOError :: String       - location
               -> Errno        - errno
               -> Maybe Handle - handle
               -> Maybe String - filename
               -> IOError

throwErrno     :: String -> IO a
```

The function `errnoToIOError` throws an `IO` exception based on the given `Errno` value. The first argument to the function should specify the location where the error occured and the third and fourth can be used to specify a file handle and filename in the course of whose's manipulation the error occured. This is optional information, which can be used to improve the accuracy of error messages. The function `throwErrno` applies `errnoToIOError` to the current value returned by `getErrno`. Its first argument specifies the location - no extra information about a file handle or filename can be provided in this case.

```
throwErrnoIf          :: (a -> Bool) -> String -> IO a       -> IO a
throwErrnoIf_         :: (a -> Bool) -> String -> IO a       -> IO ()
throwErrnoIfRetry     :: (a -> Bool) -> String -> IO a       -> IO a
throwErrnoIfRetry_    :: (a -> Bool) -> String -> IO a       -> IO ()
throwErrnoIfMinus1    :: Num a
      =>                String -> IO a       -> IO a
throwErrnoIfMinus1_   :: Num a
      =>                String -> IO a       -> IO ()
throwErrnoIfMinus1Retry
      :: Num a
      =>                String -> IO a       -> IO a
throwErrnoIfMinus1Retry_
      :: Num a
      =>                String -> IO a       -> IO ()
throwErrnoIfNull      ::                String -> IO (Ptr a) -> IO (Ptr a)
throwErrnoIfNullRetry ::                String -> IO (Ptr a) -> IO (Ptr a)
```

The function `throwErrnoIf` behaves like `throwErrno` in case that the result of the `IO` action fulfils the predicate passed as a first argument. All variants of the function whose name ends with an underscore character behave like the version without the underscore in the name, but discard the result of the `IO` action. All variants ending in "`Retry`" do retry the `IO` action when they would usually throw `eINTR` - this amounts to the standard retry loop for interrupted POSIX system calls.

Finally, `throwErrnoIfMinus1` and its variants throw the current `errno` value (or possibly retry the action) when the result of the `IO` action is `-1`, whereas `throwErrnoIfNull` does so in case of a `Ptr.nullPtr` (Section 5.29).

5.6. `CForeign`

The module `CForeign` exports the complete interface of the *C-specific* portion of the marshalling library. It includes the following modules: `CTypes` (Section 5.7), `CTypesISO` (Section 5.8), `CString` (Section 5.9), and `CError` (Section 5.5). It will usually be used in conjunction with the *language-independent* portion of the marshalling library, which is exported by the module `Foreign` (Section 5.13).

As module `CForeign` simply re-exports the interface of the other modules listed above, please refer to the documentation of these modules for the details of the provided definitions.

5.7. `CTypes`

This module is part of the language-dependent part of the Foreign Function Interface (FFI) - for the language-independent part, see Section 5.13. It defines Haskell types that can hold the primitive types of C and can directly be used in foreign import and export declarations. All definitions are together with the remainder of the C-specific marshalling support also available via the module `CForeign` (Section 5.6).

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.

5.7.1. 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
```

5.7.2. 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
```

5.8. CTypesISO

This module is part of the language-dependent part of the Foreign Function Interface (FFI) - for the language-independent part, see Section 5.13. It defines Haskell types corresponding the most important ISO types of C that are not covered in the module CTypes. All definitions are together with the remainder of the C-specific marshalling support also available via the module CForeign (Section 5.6).

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.

5.8.1. 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
```

5.8.2. 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
```

5.8.3. Misc types

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

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

5.9. CString

The module `CString` provides routines marshalling Haskell into C strings and vice versa. The marshalling takes the current Unicode encoding on the Haskell side into account. As a consequence, no guarantees can be made about the relative length of a Haskell string and its corresponding C string. Therefore, all routines provided by the present module combine memory allocation and marshalling. All definitions are together with the remainder of the C-specific marshalling support also available via the module `CForeign` (Section 5.6).

```
CString    = Ptr CChar
CStringLen = (CString, Int)
```

The library supports two string representations on the C side: the standard NUL terminated strings (`CString`) and strings with explicit length information (`CStringLen`).

```
peekCString    :: CString    -> IO String
peekCStringLen :: CStringLen -> IO String
```

The functions `peekCString` and `peekCStringLen` marshal C strings to Haskell.

```
newCString     :: String -> IO CString
newCStringLen  :: String -> IO CStringLen

withCString    :: String -> (CString    -> IO a) -> IO a
withCStringLen :: String -> (CStringLen -> IO a) -> IO a
```

The functions `newCString` and `newCStringLen` correspond to the `MarshalUtils.new` function (cf. Section 5.25). They allocate storage for the C representation of a Haskell string and also perform the marshalling. Moreover, the functions `withCString` and `withCStringLen` correspond to `MarshalUtils.with`. They allocate temporary storage passed to the functional provided in the second argument. The storage is released when the functional returns or throws an exception.

```
castCharToCChar :: Char -> CChar
castCCharToChar :: CChar -> Char
```

The two functions `castCharToCChar` and `castCCharToChar` cast Haskell characters to C characters and vice versa while ignoring the Unicode encoding of the Haskell character. These functions should be used with care.

```
data UnsafeCString   - abstract
withUnsafeCString    :: String -> (UnsafeCString -> IO a) -> IO a
```

As an *experimental* feature, we also support a variant of NUL terminated C strings whose representation is not disclosed, but which *may* lead to better performance when used in an `unsafe` foreign import. The only operation on unsafe C strings, `withUnsafeCString`, allocates temporary storage for an unsafe C string and marshalls a Haskell string into that memory area.

5.10. `DiffArray`

Diff arrays have immutable interface (Section 5.17), but rely on internal updates in place to provide fast functional update operator `(//)`.

When the `(//)` operator is applied to a diff array, its contents are physically updated in place. The old array silently changes its representation without changing the visible behavior: it stores a link to the new current array along with the difference to be applied to get the old contents.

So if a diff array is used in a single-threaded style, i.e. after `(//)` application the old version is no longer used, `a!i` takes *O(1)* time and `a // d` takes *O(`length d`)*. Accessing elements of older versions gradually becomes slower.

Updating an array which is not current makes a physical copy. The resulting array is unlinked from the old family. So you can obtain a version which is guaranteed to be current and thus have fast element access by `a // []`.

An arbitrary `MArray` type living in the `IO` monad can be converted to a diff array with the type constructor `IOToDiffArray`. It has kind `(* -> * -> *) -> * -> * -> *`, where parameters mean the following:

- internal mutable array type constructor, which is itself parametrized over index and element types
- index type
- element type

Two most important diff array types are fully polymorphic lazy boxed `DiffArray = IOToDiffArray IOArray`, and strict unboxed `DiffUArray = IOToDiffArray IOUArray`, working only for elements of primitive types, but more compact and usually faster. These type constructors should be applied to the index type and element type, providing an array type which is an instance of `IArray`.

5.11. `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).

5.11.1. 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, `TypeReps` 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 `TypeReps` 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.

5.11.2. The Typeable class

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

```
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.

5.11.3. 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
```

5.12. `Exception`

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

Exceptions are defined by the (non-abstract) `Exception` datatype. The standard Haskell `IOError` type is a synonym for `Exception`.

```
- instance of Eq, Ord, Show, Typeable
data Exception
  = IOException  IOException - IO exceptions
  | ArithException   ArithException - Arithmetic exceptions
  | ArrayException ArrayException  - Array-related exceptions
  | ErrorCall String - Calls to 'error'
  | ExitException ExitCode - Calls to 'System.exitWith'
  | 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
  | BlockedOnDeadMVar - Blocking on a dead MVar
  | NonTermination                     - Program is in an infinite loop
  | UserError           String         - General purpose exception

- 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
```

5.12.1. Kinds of exception

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

`IOException`

These are the standard IO exceptions generated by Haskell's `IO` operations.

`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.

`ExitException`

The `ExitException` exception is thrown by `System.exitWith` (and `System.exitFailure`). The `ExitCode` argument is the value passed to `System.exitWith`. An unhandled `ExitException` exception in the main thread will cause the program to be terminated with the given exit code.

`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 5.12.5.

`AsyncException`

Asynchronous exceptions. These are described in more detail in Section 5.12.7. 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 3.2), or by the system if it needs to terminate the thread for some reason.

`BlockedOnDeadMVar`

The current thread was executing a call to `takeMVar` (Section 3.4.3) 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.

`UserError`

The `UserError` exception is reserved for the user, and may be thrown with `IO.userError`.

5.12.2. Throwing exceptions

Exceptions may be thrown explicitly from anywhere:

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

From the `IO` monad, exceptions may be thrown using `ioError` (from module `IO`):

```
ioError :: IOError -> IO a
```

(recall that `IOError` is a synonym for `Exception`.

You may be wondering what the difference between these two functions is, after all the type of `ioError` is just an instance of the type of `throw`, so you could use `throw` everywhere and forget about `ioError`, right? This is mostly true, but there is one subtle difference between the two functions:

```
    throw e   `seq` return ()  ===> throw e
    ioError e `seq` return ()  ===> return ()
```

The first example will cause the exception `e` to be raised, whereas the second one won't. In fact, `ioError` will only cause an exception to be raised when it is used within the `IO` monad.

5.12.3. The `catch` functions

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

```
catch      :: IO a -> (Exception -> IO a) -> IO a
catchJust  :: (Exception -> Maybe b) -> IO a -> (b -> IO a) -> IO a
```

The simplest version is `catch`. It takes a single argument, runs it, and if an exception is raised the "handler" is executed passing the value of the exception. Otherwise, the result is returned as normal. For example:

```
    catch (openFile f ReadMode)
        (\e -> hPutStr stderr ("Couldn't open "++f++": " ++ show e))
```

For catching exceptions in pure (non-IO) expressions, the function `evaluate` is provided:

```
evaluate :: a -> IO a
evaluate a = a `seq` return a
```

`evaluate` simply forces its argument to be evaluated, then returns the result in the IO monad. This is useful with `catch`:

```
do
...
let value = read str
catch (evaluate value)
   (\e -> hPutStr stderr ("Failed to parse: " ++ show e)
```

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 `catch` execute the handler passing `ErrorCall "urk"`, or `ArithError DivideByZero`?

The answer is "either": `catch` makes a non-deterministic choice about which exception to catch. If you call it again, you might get a different exception back. This is ok, because `catch` is an IO computation.

The function `catchJust` takes 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:

```
ioErrors :: Exception -> Maybe IOError
arithExceptions  :: Exception -> Maybe ArithException
errorCalls :: Exception -> Maybe String
dynExceptions  :: Exception -> Maybe Dynamic
assertions :: Exception -> Maybe String
asyncExceptions  :: Exception -> Maybe AsyncException
userErrors   :: Exception -> Maybe String
```

For example, to catch just calls to `error` we could use something like

```
result <- catchJust errorCalls thing_to_try handler
```

Any other exceptions which aren't matched by the predicate are re-raised, and may be caught by an enclosing `catch` or `catchJust`.

NOTE: The `Prelude` also exports a function called `catch` which has the same type as `Exception.catch`, the difference being that the `Prelude` version only catches the IO and user families of exceptions (as required by Haskell 98). We recommend either hiding the `Prelude` version of `catch` when importing `Exception`, or importing `Exception` qualified, to avoid name-clashes.

5.12.4. The `try` functions

The `try` family is similar to the `catch` family:

```
try     :: IO a -> IO (Either Exception a)
tryJust :: (Exception -> Maybe b) -> IO a -> IO (Either b a)
```

The difference is that instead of executing a handler when an exception is caught, these functions return an `Either` result, with `Right a` indicating success, and `Left e` indicating that exception `e` was raised.

5.12.5. Dynamic Exceptions

Because the `Exception` datatype isn't extensible, we added an interface for throwing and catching exceptions of type `Dynamic` (see Section 5.11), which allows exception values of any type in the `Typeable` class to be thrown and caught.

```
throwDyn   :: Typeable exception => exception -> b
throwDynTo :: Typeable exception => ThreadId -> exception -> IO ()
catchDyn   :: Typeable exception => IO a -> (exception -> IO a) -> IO a
```

The `catchDyn` function only catches exceptions of the required type; all other exceptions are re-thrown as with `catchIO` and friends above.

5.12.6. Other Utilities

The `bracket` functions are useful for making sure that resources are released properly by code that may raise exceptions:

```
bracket   :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket_  :: IO a -> IO b -> IO c -> IO c
finally :: IO a -> IO b -> IO a
```

For example, to open a file, do some work on it and then close it again, we might use something like:

```
process_file =
  bracket
     (openFile "filename" ReadMode)
     (hClose)
     (\handle -> do { ... })
```

`bracket` works as follows: it executes its first argument ("open"), then its third argument, followed finally by its second argument ("close"). If the third argument happened to raise an exception, then the close operation will still be performed, and the exception will be re-raised.

This means that in the example above the file will always be closed, even if an error occurs during processing.

The arguments to `bracket` are in this order so that we can partially apply it, like:

```
withFile name = bracket (openFile name) hClose
```

The `bracket_` function is a variant of `bracket` that throws away the result of the open, and `finally` is an even simpler version where we just want some closing code.

5.12.7. Asynchronous exceptions

Asynchronous exceptions are so-called because they arise due to external influences, and can be raised at any point during execution. `StackOverflow` and `HeapOverflow` are two examples of system-generated asynchronous exceptions.

The primary source of asynchronous exceptions, however, is `throwTo`, from the `Concurrent` library (see Section 3.2):

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

`throwTo` allows one running thread to raise an arbitrary exception in another thread. The exception is therefore asynchronous with respect to the target thread, which could be doing anything at the time it receives the exception. Great care should be taken with asynchronous exceptions; it is all too easy to introduce race conditions by the over zealous use of `throwTo`.

There are two functions which allow a thread to control the delivery of asynchronous exceptions during critical sections:

```
block   :: IO () -> IO ()
unblock :: IO () -> IO ()
```

Applying `block` to a computation will execute that computation with asynchronous exceptions *blocked*. That is, any thread which attempts to raise an exception in the current thread will be blocked until asynchronous exceptions are enabled again. There's no need to worry about re-enabling asynchronous exceptions; that's done automatically on exiting the scope of `block`.

To re-enable asynchronous exceptions inside the scope of `block`, `unblock` can be used. It scopes in exactly the same way, so on exit from `unblock` asynchronous exception delivery will be disabled again.

For some examples of the use of these functions, see the definitions of `finally` and `bracket` in the sources to the `Exception` module.

5.12.7.1. Applying `block` to an exception handler

There's an implied `block` around every exception handler in a call to one of the `catch` family of functions. This is because that's what you want most of the time - it eliminates a common race condition in starting an exception handler, because there may be no exception handler on the stack to handle another exception if one arrives immediately. If asynchronous exceptions are blocked on entering the handler, though, we have time to install a new exception handler before being interrupted. If this weren't the default, you'd have to write something like

```
block (
     catch (unblock (...))
                (\e -> handler)
)
```

If you need to unblock asynchronous exceptions again in the exception handler, just use `unblock` as normal.

Note that `try` and friends *don't* have a similar default, because there is no exception handler in this case. If you want to use `try` in an asynchronous-exception-safe way, you'll need to use `block`.

5.12.7.2. Interruptible operations

Some operations are *interruptible*, which means that they can receive asynchronous exceptions even in the scope of a `block`. Any function which may itself block is defined as interruptible; this includes `takeMVar`, and most I/O-performing operations. The reason for having interruptible operations is so that we can write things like

```
block (
   a <- takeMVar m
   catch (unblock (...))
         (\e -> ...)
)
```

if the `takeMVar` wasn't interruptible, then this particular combination could lead to deadlock, because the thread itself would be blocked in a state where it can't receive any asynchronous exceptions. With `takeMVar` interruptible, however, we can be safe in the knowledge that the thread can receive exceptions right up until the point when the `takeMVar` succeeds. Similar arguments apply for other interruptible operations like `IO.openFile`.

5.13. Foreign

The Foreign Function Interface (FFI) consists of three parts:

1. foreign import and export declarations (defined in an extra document *A Haskell Foreign Function Interface*),

2. a language-independent marshalling library (whose interface is provided by the present module), and
3. a language-dependent marshalling library (see Section 5.6).

The *language-independent* portion of the marshalling library, whose entire interface is provided by the module `Foreign`, consists of the following modules: `Int` (Section 5.18), `Word` (Section 5.39), `Ptr` (Section 5.29), `ForeignPtr` (Section 5.15), `StablePtr` (Section 5.34), `Storable` (Section 5.35), `MarshalAlloc` (Section 5.22), `MarshalArray` (Section 5.23), `MarshalError` (Section 5.24), and `MarshalUtils` (Section 5.25). The *language-dependent* portion of the marshalling library is currently only available for the language C, and is essentially intended to support writing portable Haskell bindings to C libraries. All C-dependent functionality can be obtained from the module `CForeign` (Section 5.6).

The code for marshalling of Haskell structures into a foreign representation and vice versa can generally be implemented in either Haskell or the foreign language. At least if the foreign language is a significantly lower level language, such as C, there are good reasons for doing the marshalling in Haskell:

- Haskell's lazy evaluation strategy would require any foreign code that attempts to access Haskell structures to force the evaluation of the structures before accessing them. This would lead to complicated code in the foreign language, but does not need any extra consideration when coding the marshalling in Haskell.
- Despite the fact that marshalling code in Haskell tends to look like C in Haskell syntax, the strong type system still catches many errors that would otherwise lead to difficult to debug runtime faults.
- Direct access to Haskell heap structures from a language like C - especially, when marshalling from C to Haskell, i.e., when Haskell structures are created - carries the risk of corrupting the heap, which usually leads to faults that are very hard to debug. (Paradox as it may seem, the cause for corrupted C structures is usually easier to locate, at least when a conventional debugger like gdb is at hand.)

Consequently, the Haskell FFI emphasises Haskell-side marshalling.

As module `Foreign` simply re-exports the interface of the other modules listed above, please refer to the documentation of these modules for the details of the provided definitions.

5.14. `ForeignObj`

NOTE: this interface is deprecated. Please use `ForeignPtr` instead (see Section 5.15).

This module is part of the Foreign Function Interface (FFI) and will usually be imported via the module `Foreign` (see Section 5.13). The type `ForeignObj` represents references to objects that are maintained in a foreign language, i.e., that are not part of the data structures usually managed by the Haskell storage manager. The essential difference between `ForeignObj`s and vanilla memory references of type `Ptr a` (Section 5.29) is that the former may be associated with *finalisers.* A finaliser is a routine that is invoked when the Haskell storage manager detects that - within the Haskell heap and stack - there are no more references left that are pointing to the `ForeignObj`. Typically, the finaliser will, then, invoke routines in the foreign language that free the resources bound by the foreign object.

5.14.1. The Standard Interface

```
data ForeignObj - abstract handle to foreign object
instance Eq ForeignObj

newForeignObj       :: Ptr a      -> IO () -> IO ForeignObj
addForeignFinalizer :: ForeignObj -> IO () -> IO ()
withForeignObj      :: ForeignObj -> (Ptr a -> IO b) -> IO b
touchForeignObj     :: ForeignObj -> IO ()
foreignObjToPtr     :: ForeignObj -> Ptr a  - *unsafe* operation
```

The behaviour of the functions is as follows:

`newForeignObj :: Ptr a -> IO () -> IO ForeignObj`

Turns a plain memory reference into a foreign object by associating a finaliser - given by the monadic operation - with the reference. The finaliser will be executed after the last reference to the foreign object is dropped. Note that there is no guarantee on how soon the finaliser is executed after the last reference was dropped; this depends on the details of the Haskell storage manager. The only guarantee is that the finaliser runs before the program terminates.

`addForeignFinalizer :: ForeignObj -> IO () -> IO ()`

This function adds another finaliser to the given foreign object. No guarantees are made on the order in which multiple finalisers for a single object are run.

`withForeignObj :: ForeignObj -> (Ptr a -> IO b) -> IO b`

This is a way to look at the pointer living inside a foreign object. This function takes a function which is applied to that pointer. The resulting IO action is then executed. The foreign object is kept alive at least during the whole action, even if it is not used directly inside. Note that it is not safe to return the pointer from the action and use it after the action completes. All uses of the pointer should be inside the `withForeignObj` bracket.

`touchForeignObj :: ForeignObj -> IO ()`

This function ensures that the foreign object in question is alive at the given place in the sequence of IO actions. It tells that we are still interested in the object, even though we don't have anything more specific to do with a `ForeignObj` value, probably because we are playing with the address taken from it. In particular `withForeignObj` does a `touchForeignObj` after it executes the user action.

```
foreignObjToPtr :: ForeignObj -> Ptr a
```

Extract the plain memory reference contained in a foreign object. You should use `withForeignObj` instead of `foreignObjToPtr` whenever possible because the former function is safer.

This routine should be handled with a lot of care: The reference to the foreign object that is passed in a call to `foreignObjToPtr` may be the last reference to the object that exists in Haskell land. In this case, the finalisers of the foreign object may be activated any time after the call to `foreignObjToPtr` is evaluated. If the finalisers, for example, trigger deallocation of the foreign object's memory area, the `Ptr a` obtained by the call to `foreignObjToPtr` may be rendered invalid when garbage collection hits after the call. Whether this is a problem or not depends on the details of the finaliser code and the operations subsequently performed on the `Ptr a`.

`touchForeignObj` can help in ensuring that the foreign object is not finalised too early. A stable pointer (Section 5.34) can also be used to establish a guaranteed reference to the foreign object. The finalisers will, then, certainly not be run before `StablePtr.freeStablePtr` is used.

5.14.2. 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* HsForeignObj;  /* C representation of a ForeignObj */
```

5.14.3. Deprecated Functions

The following functions are deprecated in the new FFI. Do *not* use the following functions if you are interested in portability. Instead of the `indexXXX`, `readXXX`, and `writeXXX` functions, use the module `Storable` (Section 5.35).

Most of these functions are here for legacy reasons and may just vanish one day. You have been warned.

```
foreignObjToAddr :: ForeignObj -> Addr   - *unsafe* operation
writeForeignObj  :: ForeignObj -> Addr{-new value-} -> IO ()
makeForeignObj   :: Addr -> Addr -> IO ForeignObj

indexCharOffForeignObj   :: ForeignObj -> Int -> Char
indexIntOffForeignObj    :: ForeignObj -> Int -> Int
indexWordOffForeignObj   :: ForeignObj -> Int -> Word
indexAddrOffForeignObj   :: ForeignObj -> Int -> Addr
indexFloatOffForeignObj  :: ForeignObj -> Int -> Float
```

```
indexDoubleOffForeignObj :: ForeignObj -> Int -> Double
indexWord8OffForeignObj  :: ForeignObj -> Int -> Word8
indexWord16OffForeignObj :: ForeignObj -> Int -> Word16
indexWord32OffForeignObj :: ForeignObj -> Int -> Word32
indexWord64OffForeignObj :: ForeignObj -> Int -> Word64

indexInt8OffForeignObj  :: ForeignObj -> Int -> Int8
indexInt16OffForeignObj :: ForeignObj -> Int -> Int16
indexInt32OffForeignObj :: ForeignObj -> Int -> Int32
indexInt64OffForeignObj :: ForeignObj -> Int -> Int64

- read value out of mutable memory
readCharOffForeignObj    :: ForeignObj -> Int -> IO Char
readIntOffForeignObj     :: ForeignObj -> Int -> IO Int
readWordOffForeignObj    :: ForeignObj -> Int -> IO Word
readAddrOffForeignObj    :: ForeignObj -> Int -> IO Addr
readFloatOffForeignObj   :: ForeignObj -> Int -> IO Float
readDoubleOffForeignObj  :: ForeignObj -> Int -> IO Double
readWord8OffForeignObj   :: ForeignObj -> Int -> IO Word8
readWord16OffForeignObj  :: ForeignObj -> Int -> IO Word16
readWord32OffForeignObj  :: ForeignObj -> Int -> IO Word32
readWord64OffForeignObj  :: ForeignObj -> Int -> IO Word64
readInt8OffForeignObj    :: ForeignObj -> Int -> IO Int8
readInt16OffForeignObj   :: ForeignObj -> Int -> IO Int16
readInt32OffForeignObj   :: ForeignObj -> Int -> IO Int32
readInt64OffForeignObj   :: ForeignObj -> Int -> IO Int64

writeCharOffForeignObj   :: ForeignObj -> Int -> Char   -> IO ()
writeIntOffForeignObj    :: ForeignObj -> Int -> Int    -> IO ()
writeWordOffForeignObj   :: ForeignObj -> Int -> Word   -> IO ()
writeAddrOffForeignObj   :: ForeignObj -> Int -> Addr   -> IO ()
writeFloatOffForeignObj  :: ForeignObj -> Int -> Float  -> IO ()
writeDoubleOffForeignObj :: ForeignObj -> Int -> Double -> IO ()
writeWord8OffForeignObj  :: ForeignObj -> Int -> Word8  -> IO ()
writeWord16OffForeignObj :: ForeignObj -> Int -> Word16 -> IO ()
writeWord32OffForeignObj :: ForeignObj -> Int -> Word32 -> IO ()
writeWord64OffForeignObj :: ForeignObj -> Int -> Word64 -> IO ()
writeInt8OffForeignObj   :: ForeignObj -> Int -> Int8   -> IO ()
writeInt16OffForeignObj  :: ForeignObj -> Int -> Int16  -> IO ()
writeInt32OffForeignObj  :: ForeignObj -> Int -> Int32  -> IO ()
writeInt64OffForeignObj  :: ForeignObj -> Int -> Int64  -> IO ()
```

5.15. ForeignPtr

This module is part of the Foreign Function Interface (FFI) and will usually be imported via the module `Foreign` (see Section 5.13). The type `ForeignPtr` represents references to objects that are maintained in a foreign language, i.e., that are not part of the data structures usually managed by the Haskell storage manager. The essential difference between `ForeignPtr`s and vanilla memory references of type `Ptr a` (Section 5.29) is that the former may be associated with *finalisers.* A finaliser is a routine that is invoked when the Haskell storage manager detects that - within the Haskell heap and stack - there are no more references left that are pointing to the `ForeignPtr`. Typically, the finaliser will, then, invoke routines in the foreign language that free the resources bound by the foreign object.

The `ForeignPtr` is parameterised in the same way as `Ptr`. The type argument of `ForeignPtr` should normally be an instance of class `Storable` (see Section 5.35).

5.15.1. The Haskell-side Interface

```
data ForeignPtr a - abstract handle to foreign object
instance Eq (ForeignPtr a)

newForeignPtr           :: Ptr a        -> IO () -> IO (ForeignPtr a)
addForeignPtrFinalizer :: ForeignPtr a -> IO () -> IO ()
withForeignPtr          :: ForeignPtr a -> (Ptr a -> IO b) -> IO b
touchForeignPtr         :: ForeignPtr a -> IO ()
foreignPtrToPtr         :: ForeignPtr a -> Ptr a
castForeignPtr         :: ForeignPtr a -> ForeignPtr b
```

The behaviour of the functions is as follows:

`newForeignPtr :: Ptr a -> IO () -> IO (ForeignPtr a)`

Turns a plain memory reference into a foreign object by associating a finaliser - given by the monadic operation - with the reference. The finaliser will be executed after the last reference to the foreign object is dropped. Note that there is no guarantee on how soon the finaliser is executed after the last reference was dropped; this depends on the details of the Haskell storage manager. The only guarantee is that the finaliser runs before the program terminates.

`addForeignPtrFinalizer :: ForeignPtr a -> IO () -> IO ()`

This function adds another finaliser to the given foreign object. No guarantees are made on the order in which multiple finalisers for a single object are run.

`withForeignPtr :: ForeignPtr a -> (Ptr a -> IO b) -> IO b`

This is a way to look at the pointer living inside a foreign object. This function takes a function which is applied to that pointer. The resulting IO action is then executed. The foreign object is kept alive at least during the whole action, even if it is not used directly inside. Note that it is not safe to return the pointer from the action and use it after the action completes. All uses of the pointer should be inside the `withForeignPtr` bracket. The reason for this unsafety is the same as for `foreignPtrToPtr` below: the finalizer may run earlier than expected, because the compiler can only track usage of the `ForeignPtr` object, not a `Ptr` object made from it.

This function is normally used for marshalling data to or from the object pointed to by the `ForeignPtr`, using the operations from the `Storable` class.

```
foreignPtrToPtr :: ForeignPtr a -> Ptr a
```

This function extracts the pointer component of a foreign pointer. This is a potentially dangerous operations, as if the argument to `foreignPtrToPtr` is the last usage occurence of the given foreign pointer, then its finaliser(s) will be run, which potentially invalidates the plain pointer just obtained. Hence, `touchForeignPtr` must be used wherever it has to be guaranteed that the pointer lives on - i.e., has another usage occurrence.

To avoid subtle coding errors, hand written marshalling code should preferably use `withForeignPtr` rather than combinations of `foreignPtrToPtr` and `touchForeignPtr`. However, the later routines are occasionally preferred in tool generated marshalling code.

```
touchForeignPtr :: ForeignPtr a -> IO ()
```

This function ensures that the foreign object in question is alive at the given place in the sequence of IO actions. In particular `withForeignPtr` does a `touchForeignPtr` after it executes the user action.

This function can be used to express liveness dependencies between `ForeignPtr`s: for example, if the finalizer for one `ForeignPtr` touches a second `ForeignPtr`, then it is ensured that the second `ForeignPtr` will stay alive at least as long as the first. This can be useful when you want to manipulate *interior pointers* to a foreign structure: you can use `touchForeignObj` to express the requirement that the exterior pointer must not be finalized until the interior pointer is no longer referenced.

```
castForeignPtr :: ForeignPtr a -> ForeignPtr b
```

This function casts a `ForeignPtr` parameterised by one type into another type.

5.15.2. The C-side Interface

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

```
typedef void* HsForeignPtr;  /* C representation of a ForeignPtr */
```

5.16. GlaExts

The GlaExts interface provides access to extensions that only GHC implements. These currently are: unboxed types, including the representations of the primitive types (Int, Float, etc.), and the GHC primitive operations (+#, ==#, etc.).

This module used to provide access to all the Glasgow extensions, but these have since been moved into separate libraries for compatibility with Hugs (version 2.09: in fact, you can still get at this stuff via GlaExts for compatibility, but this facility will likely be removed in the future).

```
- the representation of some basic types:
data Char    = C# Char#
data Int     = I# Int#
data Addr    = A# Addr#
data Word    = W# Word#
data Float   = F# Float#
data Double  = D# Double#
data Integer = S# Int#              - small integers
             | J# Int# ByteArray#   - large integers

module GHC  - all primops and primitive types.
```

5.17. IArray

This module provides a family of immutable array data types, and is intended to be a replacement for Haskell's standard Array module. In addition, a family of mutable array types is provided by the MArray module (see Section 5.21).

5.17.1. IArray and HasBounds type classes

Two new type classes are provided, HasBounds and IArray:

```
class HasBounds a where
   bounds :: Ix ix => a ix e -> (ix,ix)

class HasBounds a => IArray a e where
   array :: Ix ix => (ix,ix) -> [(ix,e)] -> a ix e
   (!)   :: Ix ix => a ix e -> ix -> e
   (//)  :: Ix ix => a ix e -> [(ix,e)] -> a ix e

instance HasBounds (Array.Array)
instance IArray Array.Array e
```

where the type variable a denotes the array type constructor, ix denotes the index type, and e is the element type.

The Array, (!), (//) and bounds methods provided are exactly analoguous to those provided by the Haskell 98 Array module, and indeed instances of HasBounds and IArray are provided for the standard Array type.

5.17.2. UArray: immutable unboxed arrays

The UArray type is a flat, strict, unboxed array type, which has instances of IArray for common integral element types:

```
data UArray ix e
instance HasBounds UArray
instance IArray UArray Bool
instance IArray UArray Char
instance IArray UArray Int
instance IArray UArray Word
instance IArray UArray Addr
instance IArray UArray (Ptr a)
instance IArray UArray Float
instance IArray UArray Double
instance IArray UArray (StablePtr a)
instance IArray UArray Int8
instance IArray UArray Int16
instance IArray UArray Int32
instance IArray UArray Int64
instance IArray UArray Word8
instance IArray UArray Word16
instance IArray UArray Word32
instance IArray UArray Word64
```

The idea here is that any code which currently uses a standard non-strict polymorphic Array with one of these element types can be easily converted to use UArray by simply importing IArray and changing the type of the array from Array to UArray. The application will then get the performance benefit of strict unboxed arrays, whithout making significant changes to the existing code.

5.17.3. Useful combinators

The IArray module also provides versions of the standard array combinators from Haskell 98's Array module:

```
assocs     :: (Ix ix, IArray a e) => a ix e -> [(ix,e)]

indices    :: (Ix ix, HasBounds a) => a ix e -> [ix]

listArray  :: (Ix ix, IArray a e) => (ix,ix) -> [e] -> a ix e

elems      :: (Ix ix, IArray a e) => a ix e -> [e]
```

```
amap       :: (Ix ix, IArray a x, IArray a y) =>
    (x -> y) -> a ix x -> a ix y

accum      :: (Ix ix, IArray a e) =>
    (e->f->e) -> a ix e -> [(ix,f)] -> a ix e

accumArray :: (Ix ix, IArray a e) =>
    (e->f->e) -> e -> (ix,ix) -> [(ix,f)] -> a ix e

ixmap      :: (Ix ix, Ix iy) =>
   (ix,ix) -> (ix->iy) -> a iy e -> a ix e
```

5.18. Int

This interface provides a collection of sized, signed integers. The types supported are as follows:

type	number of bits
Int8	8
Int16	16
Int32	32
Int64	64

For each type *I* above, we provide the following instances.

```
data I            - Signed Ints
instance Eq       I
instance Ord      I
instance Show     I
instance Read     I
instance Bounded  I
instance Num      I
instance Real     I
instance Integral I
instance Enum     I
instance Ix       I
instance Bits     I
```

5.18.1. Notes

- All arithmetic is performed modulo 2^n.
- For coercing between various integer types, use `fromIntegral`, which is specialized for all the common cases so should be fast enough.
- The rules that hold for `Enum` instances over a bounded type such as `Int` (see the section of the Haskell report dealing with arithmetic sequences) also hold for the `Enum` instances over the various `Int` types defined here.
- Right and left shifts by amounts greater than or equal to the width of the type result in either zero or -1, depending on the sign of the value being shifted. This is contrary to the behaviour in C, which is undefined; a common interpretation is to truncate the shift count to the width of the type, for example `1 « 32 == 1` in some C implementations.
- Hugs does not provide `Int64` at the moment.

5.18.2. Deprecated Functions

The following functions are deprecated in the new FFI. Do *not* use the following functions if you are interested in portability. Most of these functions are here for legacy reasons and may just vanish one day. You have been warned.

The `Int` module also exports the overloaded operations for converting to and from Haskell `Int`s. However, for coercing between various integer types, better use `fromIntegral`, which is specialized for all the common cases so should be fast enough.

```
toInt   :: (Integral a) => a -> Int
fromInt :: (Num a) => Int -> a
```

Portability note: both Hugs98 and all releases of GHC prior to ghc-4.05 also exports these two via the Prelude. So, to have code that uses `toInt` and `fromInt` be maximally portable, make sure you add an import on `Int` (even if the version of Hugs or GHC you're currently using may not export these two from there.)

5.19. `IOExts`

This library is the home for miscellaneous IO-related extensions.

5.19.1. IO monad extensions

`fixIO :: (a -> IO a) -> IO a`

`fixIO` allows recursive IO operations to be defined. The first argument to `fixIO` should be a function that takes its own output as an argument (sometimes called "tying the knot").

`unsafePerformIO :: IO a -> a`

This is the "back door" into the `IO` monad, allowing `IO` computation to be performed at any time. For this to be safe, the `IO` computation should be free of side effects and independent of its environment.

If the I/O computation wrapped in `unsafePerformIO` performs side effects, then the relative order in which those side effects take place (relative to the main I/O trunk, or other calls to `unsafePerformIO`) is indeterminate.

However, it is less well known that `unsafePerformIO` is not type safe. For example:

```
test :: IORef [a]
test = unsafePerformIO $ newIORef []

main = do
        writeIORef test [42]
        bang <- readIORef test
        print (bang :: [Char])
```

This program will core dump. This problem with polymorphic references is well known in the ML community, and does not arise with normal monadic use of references. There is no easy way to make it impossible once you use `unsafePerformIO`. Indeed, it is possible to write `coerce :: a -> b` with the help of `unsafePerformIO`. So be careful!

`unsafeInterleaveIO :: IO a -> IO a`

`unsafeInterleaveIO` allows `IO` computation to be deferred lazily. When passed a value of type `IO a`, the `IO` will only be performed when the value of the `a` is demanded. This is used to implement lazy file reading, see `IO.hGetContents`.

5.19.2. Mutable Variables

```
data IORef      - instance of: Eq
newIORef        :: a -> IO (IORef a)
readIORef       :: IORef a -> IO a
writeIORef      :: IORef a -> a -> IO ()
modifyIORef     :: IORef a -> (a -> a) -> IO ()
mkWeakIORef     :: IORef a -> IO () -> IO (Weak (IORef a))

- deprecated, use modifyIORef
updateIORef     :: IORef a -> (a -> a) -> IO ()
```

5.19.3. Mutable Arrays

```
data IOArray          -- instance of: Eq
newIOArray            :: Ix ix => (ix,ix) -> elt -> IO (IOArray ix elt)
boundsIOArray       :: Ix ix => IOArray ix elt -> (ix, ix)
readIOArray         :: Ix ix => IOArray ix elt -> ix -> IO elt
writeIOArray        :: Ix ix => IOArray ix elt -> ix -> elt -> IO ()
freezeIOArray       :: Ix ix => IOArray ix elt -> IO (Array ix elt)
thawIOArray         :: Ix ix => Array ix elt -> IO (IOArray ix elt)
unsafeFreezeIOArray :: Ix ix => IOArray ix elt -> IO (Array ix elt)
unsafeThawIOArray   :: Ix ix => Array ix elt -> IO (IOArray ix elt)
```

Note: `unsafeFreezeIOArray` and `unsafeThawIOArray` are not provided by Hugs.

5.19.4. Extended file modes

```
data IOModeEx
 = BinaryMode IOMode
 | TextMode   IOMode
   deriving (Eq, Read, Show)

openFileEx         :: FilePath -> IOModeEx -> IO Handle
hSetBinaryMode     :: Handle -> Bool -> IO Bool
```

GHC's implementation of the IO library distinguishes between binary- and text-mode files. This unfortunate hack is imposed on us by the need to support Win32 platforms.

On Win32, files opened in text mode are subject to CR-LF translation. When reading a handle in text mode, CR-LF sequences in the physical file are translated into lone LFs in the stream presented to the Haskell program. Writes to a text mode handle are subject to the inverse transformation.

On Unix platforms there is no such translation. What you get is exactly the contents of the file, and vice versa.

Unfortunately this behaviour makes it difficult to correctly implement file-positioning operations in text mode on Win32. If you want to use such operations, you must first place the handle in binary mode. Failure to do so results in IO exceptions being raised. This applies only to Win32, and not to any other platforms. If your programs use seek operations and you want them to be portable between Unix and Win32, you need to ensure the relevant handles are in binary mode.

You can get hold of a binary-mode file handle one of two ways. Either open the file with `openFileEx`, which allows the mode to be specified. Or, if you already have an open handle, use `hSetBinaryMode` to change its mode.

Also as a result of this, note that on Win32 there are also several operations which, whist still allowed, may give different results in text mode than their Unix counterparts. These are: changing buffering modes of a handle (`hSetBuffering`), and writing to a read-write handle. In both cases, the read-buffer associated with the handle needs to be flushed, and, due to the Win32 text mode translation, the resulting physical file position following the flush may be wrong.

This issue of seeking in the presence of a non-identity transform between file and buffer contents will need to be revisited when the library is re-done to properly support Unicode. The present arrangement is the least-worst kludge we could come up with at present.

5.19.5. Bulk transfers

```
hGetBuf       :: Handle -> Addr -> Int -> IO Int
hPutBuf       :: Handle -> Addr -> Int -> IO ()
```

These functions read and write chunks of data to/from a handle. They will return only when either the full buffer has been transfered, or the end of file is reached (in the case of `hGetBuf`.

```
hGetBufBA     :: Handle -> MutableByteArray RealWorld a -> Int -> IO Int
hPutBufBA     :: Handle -> MutableByteArray RealWorld a -> Int -> IO ()
```

These functions mirror the previous two functions, but operate on `MutableByteArray`s instead of `Addr`s. This may be more convenient and/or faster, depending on the circumstances.

5.19.6. Terminal control

```
hIsTerminalDevice :: Handle -> IO Bool
hSetEcho          :: Handle -> Bool -> IO ()
hGetEcho          :: Handle -> IO Bool
```

5.19.7. Redirecting handles

```
withHandleFor :: Handle -> Handle -> IO a -> IO a
withStdout    :: Handle -> IO a -> IO a
withStdin     :: Handle -> IO a -> IO a
withStderr    :: Handle -> IO a -> IO a
```

5.19.8. Trace

```
trace :: String -> a -> a
```

When called, `trace` prints the string in its first argument to standard error, before returning the second argument as its result. The `trace` function is not referentially transparent, and should only be used for debugging, or for monitoring execution. Some implementations of `trace` may decorate the string that's output to indicate that you're tracing.

`trace` is implemented using `unsafePerformIO`.

5.19.9. Extra `IOError` Predicates

The `IO` module provides several predicates over the `IOError` type, such as `isEOFError`, `isDoesNotExistError`, and so on. Here we define an extended set of these predicates, taking into account more types of error:

```
isHardwareFault    :: IOError -> Bool
isInappropriateType    :: IOError -> Bool
isInterrupted    :: IOError -> Bool
isInvalidArgument    :: IOError -> Bool
isOtherError    :: IOError -> Bool
isProtocolError    :: IOError -> Bool
isResourceVanished    :: IOError -> Bool
isSystemError    :: IOError -> Bool
isTimeExpired    :: IOError -> Bool
isUnsatisfiedConstraints :: IOError -> Bool
isUnsupportedOperation   :: IOError -> Bool
isComError               :: IOError -> Bool  - Win32 only
```

5.19.10. Miscellany

```
unsafePtrEq            :: a -> a -> Bool
slurpFile              :: FilePath -> IO (Addr, Int)
hConnectTo             :: Handle -> Handle -> IO ()
performGC              :: IO ()
freeHaskellFunctionPtr :: Addr -> IO ()
```

`performGC` triggers an immediate garbage collection

`unsafePtrEq` compares two values for pointer equality without evaluating them. The results are not referentially transparent and may vary significantly from one compiler to another or in the face of semantics-preserving program changes. However, pointer equality is useful in creating a number of referentially transparent constructs such as this simplified memoisation function:

```
> cache :: (a -> b) -> (a -> b)
> cache f = \x -> unsafePerformIO (check x)
>  where
>   ref = unsafePerformIO (newIORef (error "cache", error "cache"))
>   check x = readIORef ref »= \ (x’,a) ->
>          if x ‘unsafePtrEq‘ x’ then
>            return a
```

```
>           else
>             let a = f x in
>             writeIORef ref (x, a) »
>             return a
```

5.20. LazyST

This library is identical to `ST` except that the `ST` monad instance is *lazy*. The lazy ST monad tends to be more prone to space leaks than the strict version, so most programmers will use the former unless laziness is explicitly required. `LazyST` provides two additional operations:

```
lazyToStrictST :: LazyST.ST s a -> ST.ST s a
strictToLazyST :: ST.ST s a -> LazyST.ST s a
```

These are used to convert between lazy and strict state threads. The semantics with respect to laziness are as you would expect: the strict state thread passed to `strictToLazyST` is not performed until the result of the lazy state thread it returns is demanded.

5.21. MArray

The `MArray` module provides a class of mutable arrays, parameterised over the array type, element type and the monad in which the array can be used:

```
  class (Monad m, HasBounds a) => MArray a e m where
     newArray   :: Ix ix => (ix,ix) -> e -> m (a ix e)
     newArray_  :: Ix ix => (ix,ix) -> m (a ix e)
     readArray  :: Ix ix => a ix e -> ix -> m e
     writeArray :: Ix ix => a ix e -> ix -> e -> m ()
```

There are two functions for building arrays: `newArray` and `newArray_`. The former takes an initial value and sets every element of the new array to this value, whereas the latter leaves all of the elements of the new array undefined.

The `readArray` and `writeArray` operations allow for reading and writing to/from the array.

The `IOArray` (Section 5.19.3) and `STArray` (Section 5.31) types are both instances of `MArray`:

```
data STArray s ix e
instance HasBounds (STArray s)
instance MArray (STArray s) e (ST s)

data IOArray ix e
instance HasBounds IOArray
```

```
instance MArray IOArray e IO
```

There are also strict unboxed versions of `IOArray` and `STArray`, namely `IOUArray` `STUArray` and instances of `MArray` for these types with common integral element types:

```
data STUArray s ix e
instance HasBounds (STUArray s)
instance MArray (STArray s) Bool          (ST s)
instance MArray (STArray s) Char          (ST s)
instance MArray (STArray s) Int           (ST s)
instance MArray (STArray s) Word          (ST s)
instance MArray (STArray s) Addr          (ST s)
instance MArray (STArray s) (Ptr a)       (ST s)
instance MArray (STArray s) Float         (ST s)
instance MArray (STArray s) Double        (ST s)
instance MArray (STArray s) (StablePtr a) (ST s)
instance MArray (STArray s) Int8          (ST s)
instance MArray (STArray s) Int16         (ST s)
instance MArray (STArray s) Int32         (ST s)
instance MArray (STArray s) Int64         (ST s)
instance MArray (STArray s) Word8         (ST s)
instance MArray (STArray s) Word16        (ST s)
instance MArray (STArray s) Word32        (ST s)
instance MArray (STArray s) Word64        (ST s)

data IOUArray ix e
instance HasBounds IOUArray
instance MArray IOArray Bool          IO
instance MArray IOArray Char          IO
instance MArray IOArray Int           IO
instance MArray IOArray Word          IO
instance MArray IOArray Addr          IO
instance MArray IOArray (Ptr a)       IO
instance MArray IOArray Float         IO
instance MArray IOArray Double        IO
instance MArray IOArray (StablePtr a) IO
instance MArray IOArray Int8          IO
instance MArray IOArray Int16         IO
instance MArray IOArray Int32         IO
instance MArray IOArray Int64         IO
instance MArray IOArray Word8         IO
instance MArray IOArray Word16        IO
instance MArray IOArray Word32        IO
instance MArray IOArray Word64        IO
```

5.21.1. Freezing and thawing arrays

An `MArray` can be converted into an `IArray` (Section 5.17), by *freezing* it:

```
freeze :: (Ix ix, MArray a e m, IArray b e) => a ix e -> m (b ix e)
```

The `IArray` returned is independent of the original `MArray`, so further modifications to the mutable version won't affect the frozen one. The usual implementation of `freeze` is to make a copy of the array.

In many cases, the additional copy is unnecessary, for example when freezing is the last operation on the mutable version of the array. For these cases, an unsafe version of `freeze` is provided which may not copy[5] the array:

```
unsafeFreeze :: (Ix ix, MArray a e m, IArray b e) => a ix e -
> m (b ix e)
```

The dual to `freeze` is `thaw`, which converts an `IArray` into an `MArray`:

```
thaw :: (Ix ix, IArray a e, MArray b e m) => a ix e -> m (b ix e)
```

As with `freeze`, `thaw` needs to make a copy of the array. For the cases when you know the copy is unnecessary, we also provide `unsafeThaw`:

```
unsafeThaw :: (Ix ix, IArray a e, MArray b e m) => a ix e -> m (b ix e)
```

5.21.2. Useful combinators

`MArray` also provides versions of the following standard array combinators. Note that they have slightly different types from the versions in `Array` and `IArray`, mostly to accomodate the requirement that an `MArray` must be used within a monad:

```
getAssocs    :: (Ix ix, MArray a e m) => a ix e -> m [(ix,e)]

indices      :: (Ix ix, HasBounds a) => a ix e -> [ix]

mapArray     :: (Ix ix, MArray a x m, MArray a y m) =>
                    (x->y) -> a ix x -> m (a ix y)

newListArray :: (Ix ix, MArray a e m) => (ix,ix) -> [e] -> m (a ix e)

getElems     :: (Ix ix, MArray a e m) => a ix e -> m [e]

mapIndixes   :: (Ix ix, Ix iy, MArray a e m) =>
                    (ix,ix) -> (ix->iy) -> a iy e -> m (a ix e)
```

5.22. `MarshalAlloc`

The module `MarshalAlloc` provides operations to allocate and deallocate blocks of *raw memory* (i.e., unstructured chunks of memory outside of the area maintained by the Haskell storage manager). These memory blocks are commonly used to pass compound data structures to foreign functions or to provide space in which compound result values are obtained from foreign functions. For example, Haskell lists are typically passed as C arrays to C functions; the storage space for such an array can be allocated by functions (such as `mallocBytes`) contained in the present module.

```
malloc       :: Storable a =>        IO (Ptr a)
mallocBytes  ::               Int -> IO (Ptr a)

alloca       :: Storable a =>        (Ptr a -> IO b) -> IO b
allocaBytes  ::               Int -> (Ptr a -> IO b) -> IO b

reallocBytes :: Ptr a -> Int -> IO (Ptr a)

free         :: Ptr a -> IO ()
```

The functions `mallocBytes`, `reallocBytes`, and `free` correspond to the standard C functions `malloc()`, `realloc()`, and `free()`, respectively. The function `malloc` essentially behaves like `mallocBytes`, but allocates a block of memory whose size is determined by `Storable.sizeOf`. The two functions `alloca` and `allocaBytes` allocate memory as `malloc` and `mallocBytes` do, but they pass a pointer to the allocate area to the functional passed as an argument and free the storage upon completion of that functional. If the Haskell system supports exceptions, the `alloca` family of functions is exception-safe, i.e., allocated memory is freed if an exception is thrown in the marshalling functional.

5.23. MarshalArray

The module `MarshalArray` provides operations for marshalling Haskell lists into monolithic arrays and vice versa. Most functions come in two flavours: one for arrays terminated by a special termination element and one where an explicit length parameter is used to determine the extent of an array. The typical example for the former case are C's NUL terminated strings. However, please note that C strings should usually be marshalled with the functions provided by `CString` (Section 5.9) as these functions take care of the Unicode encoding, which is ignored by the functions in the present module. All functions specifically operating on arrays that are terminated by a special termination element have a name ending on `0` - e.g., `mallocArray` allocates space for an array of the given size, whereas `mallocArray0` allows for one more element.

The following six functions allocate storage for arrays.

```
mallocArray    :: Storable a => Int -> IO (Ptr a)
mallocArray0   :: Storable a => Int -> IO (Ptr a)

allocaArray    :: Storable a => Int -> (Ptr a -> IO b) -> IO b
allocaArray0   :: Storable a => Int -> (Ptr a -> IO b) -> IO b
```

```
reallocArray   :: Storable a => Ptr a -> Int -> IO (Ptr a)
reallocArray0  :: Storable a => Ptr a -> Int -> IO (Ptr a)
```

The semantics of `mallocArray`, `allocaArray`, and `reallocArray` is like that of the functions `malloc`, `alloca`, and `realloc` provided by the module `MarshalAlloc` (Section 5.24), respectively, except that they allocate enough storage for a sequence of elements whose length is explicitly given. The functions `mallocArray0`, `allocaArray0`, and `reallocArray0` allocate storage for one additional element to allow for a termination indicator.

The next four functions marshal lists into arrays and vice versa.

```
peekArray  :: Storable a =>         Int -> Ptr a -> IO [a]
peekArray0 :: (Storable a, Eq a) => a   -> Ptr a -> IO [a]

pokeArray  :: Storable a =>      Ptr a -> [a] -> IO ()
pokeArray0 :: Storable a => a -> Ptr a -> [a] -> IO ()
```

The functions `peekArray` and `peekArray0` correspond to `Storable.peek` and the functions `pokeArray` and `pokeArray0` correspond to `Storable.poke` (cf. Section 5.35). The first argument of `peekArray0` and `pokeArray0` is the element value indicating termination of the array. This element will not be included in the Haskell list representation of the marshalled value.

The next four functions combine allocation and marshalling.

```
newArray       :: Storable a =>      [a] -> IO (Ptr a)
newArray0      :: Storable a => a -> [a] -> IO (Ptr a)

withArray      :: Storable a =>      [a] -> (Ptr a -> IO b) -> IO b
withArray0     :: Storable a => a -> [a] -> (Ptr a -> IO b) -> IO b

destructArray  :: Storable a => Int -> Ptr a -> IO ()
destructArray0 :: (Storable a, Eq a) => a -> Ptr a -> IO ()
```

The functions `newArray` and `newArray0` correspond to `MarshalUtils.new`, the functions `withArray` and `withArray0` correspond to `MarshalUtils.with`, and the functions `destructArray` and `destructArray0` correspond to `Storable.destruct` (cf. Section 5.25, Section 5.35).

```
copyArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()
moveArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()
```

The two functions `copyArray` and `moveArray` copy entire arrays and behave like the routines `MarshalUtils.copyBytes` and `MarshalUtils.moveBytes`, respectively. In particular, `moveArray` allows the source and destination array to overlap, whereas `copyArray` does *not* allow overlapping arrays. Both functions take a reference to the destination array as their first, and a reference to the source as their second argument. However, the count argument here is in number of array elements (whose type is specified by the parametrised pointer arguments) instead of number of bytes.

```
lengthArray0 :: (Storable a, Eq a) => a -> Ptr a -> IO Int
```

`lengthArray0` finds the length of an array whose end is marked by a distinguished value. The length is measured in array elements (not bytes) and doesn't include the terminator.

```
advancePtr :: Storable a => Ptr a -> Int -> Ptr a
```

Finally, `advancePtr`, given a reference to an array, advances that reference by as many array elements (not bytes) as specified.

5.24. MarshalError

The module `MarshalError` provides language independent routines for converting error conditions of external functions into Haskell `IO` monad exceptions.

```
throwIf      :: (a -> Bool) -> (a -> String) -> IO a       -> IO a
throwIf_     :: (a -> Bool) -> (a -> String) -> IO a       -> IO ()
throwIfNeg   :: (Ord a, Num a)
    =>                 (a -> String) -> IO a       -> IO a
throwIfNeg_ :: (Ord a, Num a)
    =>                 (a -> String) -> IO a       -> IO ()
throwIfNull ::                 String        -> IO (Ptr a) -> IO (Ptr a)

void         :: IO a -> IO ()
```

The first two arguments to `throwIf` are a predicate yielding `True` if an error condition is met and a function that computes from that error condition an error message, which will be included in the raised `IO` exception. The third argument is an `IO` action whose result will be tested for the aforementioned error condition. If no error occurs, the result of the action is the result of the whole operation. The variant `throwIf_` works exactly in the same way, but discards the result of the `IO` action in any case.

The two functions `throwIfNeg` and `throwIfNeg_` instantiate `throwIf` and `throwIf_` with a predicate testing for a negative result. Similarly, `throwIfNull` instantiates `throwIf` with a predicate testing for a `Ptr.nullPtr` (Section 5.29).

Finally, the function `void` discards the result of an `IO` action.

5.25. MarshalUtils

The module `MarshalUtils` contains a set of useful utility routines for marshalling.

The following two routines combine storage allocation and marshalling for types in the `Storable` class (Section 5.35).

```
new  :: Storable a => a -> IO (Ptr a)
with :: Storable a => a -> (Ptr a -> IO b) -> IO b
```

The function `new` allocates storage for its argument using `MarshalAlloc.malloc` (Section 5.22), and then, stores its argument into the newly allocated area. The function `with` works similarly, but passes the newly allocated memory area to the functional given as the second argument. The object is destructed and the memory area is freed upon return of the functional. This happens even if the functional throws an exception.

GHC-specific implementation note: The function `with` is currently called `withObject`.

```
fromBool :: Num a => Bool -> a
toBool   :: Num a => a -> Bool
```

The functions `fromBool` and `toBool` are conversions between Haskell Boolean values and numeric representations of Boolean values, where `False` is represented by `0` and `True` by any non-zero value.

```
maybeNew  :: (      a -> IO (Ptr a))
  -> (Maybe a -> IO (Ptr a))
maybeWith :: (      a -> (Ptr b -> IO c) -> IO c)
  -> (Maybe a -> (Ptr b -> IO c) -> IO c)
maybePeek :: (Ptr a -> IO       b )
  -> (Ptr a -> IO (Maybe b))
```

The function `maybeNew` transforms a combined allocation and marshalling combinator such that it accepts a value wrapped into the `Maybe` type. In case of, `Nothing`, no storage is allocated and `Ptr.nullPtr` returned (Section 5.29). The function `maybeWith` does the same for combined allocation and marshalling combinators bracketing a functional.

Finally, `maybePeek` transforms a `peek` combinator such that it yields `Nothing` if passed a `Ptr.nullPtr`.

```
withMany :: (a -> (b -> res) -> res) -> [a] -> ([b] -> res) -> res
```

Given a combined allocation and marshalling combinator, the function `withMany` applies it to all elements of a list of values that have to be marshalled individually.

```
copyBytes :: Ptr a -> Ptr a -> Int -> IO ()
moveBytes :: Ptr a -> Ptr a -> Int -> IO ()
```

The two functions `copyBytes` `moveBytes` are Haskell variants of the standard C library routines `memcpy()` and `memmove()`, respectively. As with their C counterparts, `moveBytes` allows the source and destination array to overlap, whereas `copyBytes` does *not* allow overlapping areas. Both functions take a reference to the destination area as their first, and a reference to the source as their second argument - i.e., the argument order is as in an assignment.

5.26. MutableArray

NOTE: The `MutableArray` interface is deprecated, please use `MArray` (Section 5.21) instead.

The `MutableArray` interface provide operations for reading and writing values to mutable arrays. There's two kinds of mutable arrays, the mutatable version of Haskell `Array`s and *mutable byte arrays*, chunks of memory containing values of some basic type.

5.26.1. Mutable arrays

The mutable array section of the API provides the following operations:

```
- mutable arrays:
newArray      :: Ix ix -> (ix,ix) -> elt -> ST s (MutableArray s ix elt)
boundsOfArray :: Ix ix => MutableArray s ix elt -> (ix, ix)
readArray     :: Ix ix => MutableArray s ix elt -> ix -> ST s elt
writeArray    :: Ix ix => MutableArray s ix elt -> ix -> elt -> ST s ()
freezeArray   :: Ix ix => MutableArray s ix elt -> ST s (Array ix elt)
thawArray     :: Ix ix => Array ix elt -> ST s (MutableArray s ix elt)

unsafeFreezeArray   :: Ix ix => MutableArray s ix elt -
> ST s (Array ix elt)
unsafeThawArray     :: Ix ix => Array ix elt -
> ST s (MutableArray s ix elt)
```

Remarks:

- The `freezeArray` action converts a mutable array into an immutable one by copying, whereas `unsafeFreezeArray` returns an immutable array that is effectively just the type cast version of the mutable array. Should you write to the mutable array after it has been (unsafely) frozen, you'll side-effect the immutable array in the process. Please don't :-)
- The operation `thawArray` goes the other way, converting an immutable `Array` into a mutable one. This is done by copying. The operation `unsafeThawArray` is also provided, which places the same kind of proof obligation on the programmer as `unsafeFreezeArray` does.

5.26.2. Mutable byte arrays

```
- creators:
newCharArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newAddrArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newIntArray       :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newWordArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newFloatArray     :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
```

```
newDoubleArray    :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newStablePtrArray :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)

boundsOfMutableByteArray
                   :: Ix ix => MutableByteArray s ix -> (ix, ix)

readCharArray      :: Ix ix => MutableByteArray s ix -> ix -> ST s Char
readIntArray       :: Ix ix => MutableByteArray s ix -> ix -> ST s Int
readAddrArray      :: Ix ix => MutableByteArray s ix -> ix -> ST s Addr
readFloatArray     :: Ix ix => MutableByteArray s ix -> ix -> ST s Float
readDoubleArray    :: Ix ix => MutableByteArray s ix -> ix -> ST s Double
readStablePtrArray :: Ix ix => MutableByteArray s ix -> ix -
> ST s (StablePtr a)
readWord8Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word8
readWord16Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word16
readWord32Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word32
readWord64Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word64
readInt8Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Int8
readInt16Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Int16
readInt32Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Int32
readInt64Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Int64

writeCharArray        :: Ix ix => MutableByteArray s ix -> ix -> Char -
> ST s ()
writeIntArray         :: Ix ix => MutableByteArray s ix -> ix -> Int  -
> ST s ()
writeAddrArray        :: Ix ix => MutableByteArray s ix -> ix -> Addr -
> ST s ()
writeFloatArray       :: Ix ix => MutableByteArray s ix -> ix -> Float -
> ST s ()
writeDoubleArray      :: Ix ix => MutableByteArray s ix -> ix -> Double -
> ST s ()
writeStablePtrArray   :: Ix ix => MutableByteArray s ix -> ix -
> StablePtr a -> ST s ()
writeWord8Array
:: Ix ix => MutableByteArray s ix -> ix -> Word8  -> ST s ()
writeWord16Array      :: Ix ix => MutableByteArray s ix -> ix -> Word16 -
> ST s ()
writeWord32Array      :: Ix ix => MutableByteArray s ix -> ix -> Word32 -
> ST s ()
writeWord64Array      :: Ix ix => MutableByteArray s ix -> ix -> Word64 -
> ST s ()
writeInt8Array
:: Ix ix => MutableByteArray s ix -> ix -> Int8  -> ST s ()
writeInt16Array       :: Ix ix => MutableByteArray s ix -> ix -> Int16 -
> ST s ()
writeInt32Array
:: Ix ix => MutableByteArray s ix -> ix -> Int32 -> ST s ()
```

```
writeInt64Array
:: Ix ix => MutableByteArray s ix -> ix -> Int64 -> ST s ()

freezeByteArray       :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)
unsafeFreezeByteArray  :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)

sizeofMutableByteArray :: Ix ix => MutableByteArray s ix -> Int

thawByteArray       :: Ix ix => ByteArray ixt -
> ST s (MutableByteArray s ix)
unsafeThawByteArray :: Ix ix => ByteArray ixt -
> ST s (MutableByteArray s ix)
```

Remarks:

- A Mutable byte array is created by specifying its size in units of some basic type. For example,

  ```
  mkPair :: ST s (MutableByteArray s Int)
  mkPair = newIntArray (0,1)
  ```

creates a mutable array capable of storing two `Int`s. Notice that the range size *is not in bytes*, but in units of the basic type.

- A mutable byte array is not parameterised over the kind of values it contains. A consequence of this is that it is possible to have byte arrays containing a mix of basic types, or even read a value from the array at a different type from which it was written, e.g.,

  ```
  isLitteEndian :: IO Bool
  isLitteEndian = stToIO $ do
     x <- newIntArray (0,1)
     writeIntArray x 1
     v <- readCharArray x 0
     return (v == chr 1)
  ```

It's left as an exercise for the reader to determine whether having byte arrays not be parameterised over the type of values they contain is a bug or a feature..

- As for mutable arrays, operations for turning mutable byte arrays into immutable byte arrays are also provided by the `freeze*` class of actions. There's also the non-copying `unsafeFreezeByteArray`.
- Operations for going the other way, where an immutable byte array is 'thawed' are also provided. `thawByteArray` does this by copying, whereas `unsafeThawByteArray` does not
- The operation `sizeofMutableByteArray` returns the size of the array, *in bytes.*

5.27. NumExts

The NumExts interface collect together various numeric operations that have proven to be commonly useful

```
 - Going between Doubles and Floats:
doubleToFloat :: Double -> Float
floatToDouble :: Float  -> Double

showHex       :: Integral a => a -> ShowS
showOct       :: Integral a => a -> ShowS
showBin       :: Integral a => a -> ShowS

showIntAtBase :: Integral a
      => a            - base
      -> (a -> Char)  - digit to char
      -> a            - number to show.
      -> ShowS

showListWith  :: (a -> ShowS) -> [a] -> ShowS
```

Notes:

- If doubleToFloat is applied to a Double that is within the representable range for Float, the result may be the next higher or lower representable Float value. If the Double is out of range, the result is undefined.
- No loss of precision occurs in the other direction with floatToDouble, the floating value remains unchanged.
- showOct, showHex and showBin will prefix 0o, 0x and 0b, respectively. Like Numeric.showInt, these show functions work on positive numbers only.
- showIntAtBase is the more general function for converting a number at some base into a series of characters. The above show* functions use it, for instance, here's how showHex could be defined

  ```
  showHex :: Integral a => a -> ShowS
  showHex n r =
   showString "0x" $
   showIntAtBase 16 (toChrHex) n r
   where
    toChrHex d
      | d < 10    = chr (ord '0' + fromIntegral d)
      | otherwise = chr (ord 'a' + fromIntegral (d - 10))
  ```

- showListWith is strictly speaking not a 'NumExts' kind of function, but it's sometimes useful in conjunction with the other show* functions that NumExts exports. It is the non-overloaded version of showList, allowing you to supply the shows function to use per list element. For instance,

```
putStrLn (NumExts.showListWith NumExts.showHex [0..16])
```

will print out the elements of `[0..16]` in hexadecimal form.

5.28. PackedString

The basic type and functions available are:

```
data PackedString - abstract

packString          :: [Char] -> PackedString
packStringST        :: [Char] -> ST s PackedString
packCBytesST        :: Int -> Addr -> ST s PackedString
packBytesForCST     :: [Char] -> ST s (ByteArray Int)
byteArrayToPS       :: ByteArray Int -> PackedString
unsafeByteArrayToPS :: ByteArray a   -> Int -> PackedString
psToByteArray       :: PackedString -> ByteArray Int
psToByteArrayST     :: PackedString -> ST s (ByteArray Int)

unpackPS        :: PackedString -> [Char]
```

We also provide a wad of list-manipulation-like functions:

```
nilPS       :: PackedString
consPS      :: Char -> PackedString -> PackedString

headPS      :: PackedString -> Char
tailPS      :: PackedString -> PackedString
nullPS      :: PackedString -> Bool
appendPS    :: PackedString -> PackedString -> PackedString
lengthPS    :: PackedString -> Int
indexPS     :: PackedString -> Int -> Char
            - 0-origin indexing into the string
mapPS       :: (Char -> Char) -> PackedString -> PackedString
filterPS    :: (Char -> Bool) -> PackedString -> PackedString
foldlPS     :: (a -> Char -> a) -> a -> PackedString -> a
foldrPS     :: (Char -> a -> a) -> a -> PackedString -> a
takePS      :: Int -> PackedString -> PackedString
dropPS      :: Int -> PackedString -> PackedString
splitAtPS   :: Int -> PackedString -> (PackedString, PackedString)
takeWhilePS :: (Char -> Bool) -> PackedString -> PackedString
dropWhilePS :: (Char -> Bool) -> PackedString -> PackedString
spanPS      :: (Char -> Bool) -> PackedString -
> (PackedString, PackedString)
breakPS     :: (Char -> Bool) -> PackedString -
> (PackedString, PackedString)
```

```
linesPS     :: PackedString -> [PackedString]
wordsPS     :: PackedString -> [PackedString]
reversePS   :: PackedString -> PackedString
concatPS    :: [PackedString] -> PackedString
elemPS      :: Char -> PackedString -> Bool
  - Perl-style split&join
splitPS     :: Char -> PackedString -> [PackedString]
splitWithPS :: (Char -> Bool) -> PackedString -> [PackedString]
joinPS      :: PackedString -> [PackedString] -> PackedString

substrPS   :: PackedString -> Int -> Int -> PackedString
           - pluck out a piece of a PackedString
           - start and end chars you want; both 0-origin-specified
```

5.29. Ptr

The module `Ptr` provides typed pointers to foreign data. It is part of the Foreign Function Interface (FFI) and will normally be imported via the module `Foreign` (see Section 5.13).

5.29.1. Data Pointers

```
data Ptr a - abstract handle for pointers to data
- instance of: Eq, Ord, Show

nullPtr  :: Ptr a
castPtr  :: Ptr a -> Ptr b
plusPtr  :: Ptr a -> Int -> Ptr b
alignPtr :: Ptr a -> Int -> Ptr a
minusPtr :: Ptr a -> Ptr b -> Int
```

The following specifies the semantics of the data definition and the functions.

`data Ptr a`

A value of type `Ptr a` represents a pointer to an object, or an array of objects, which may be marshalled to or from Haskell values of type `a`. The type `a` will normally be an instance of class `Storable` (see Section 5.35), which provides the marshalling operations.

`nullPtr :: Ptr a`

The constant `nullPtr` contains a distinguished value of `Ptr` that is not associated with a valid memory location.

```
castPtr :: Ptr a -> Ptr b
```

The `castPtr` function casts a pointer from one type to another.

```
plusPtr :: Ptr a -> Int -> Ptr b
```

Advances the given address by the given offset in bytes.

```
alignPtr :: Ptr a -> Int -> Ptr a
```

Given an arbitrary address and an alignment constraint, `alignPtr` 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.

```
minusPtr :: Ptr a -> Ptr b -> Int
```

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

```
p2 == p1 `plusPtr` (p2 `minusPtr` p1)
```

5.29.2. Function Pointers

```
data FunPtr a - abstract handle for pointers to functions
- instance of: Eq, Ord, Show

nullFunPtr        :: FunPtr a
castFunPtr        :: FunPtr a -> FunPtr b
freeHaskellFunPtr :: FunPtr a -> IO ()
castFunPtrToPtr   :: FunPtr a -> Ptr b    - see warning below
castPtrToFunPtr   :: Ptr a -> FunPtr b    - see warning below
```

```
data FunPtr
```

A value of type `FunPtr a` is a pointer to a piece of code. It may be the pointer to a C function or to a Haskell function created using `foreign export dynamic`. A `foreign export dynamic` should normally be declared to produce a `FunPtr` of the correct type. For example:

```
type Compare = Int -> Int -> Bool
foreign export dynamic mkCompare :: Compare -> IO (FunPtr Compare)
```

```
nullFunPtr :: FunPtr a
```

The constant `nullFunPtr` contains a distinguished value of `Ptr` that is not associated with a valid memory location.

```
castFunPtr :: FunPtr a -> FunPtr b
```

Casts a `FunPtr` to a `FunPtr` of a different type.

`freeHaskellFunPtr :: FunPtr a -> IO ()`

Releases the storage associated with the given `FunPtr`. This should be called whenever the return value from a `foreign export dynamic` function is no longer required, otherwise the storage it uses will leak.

`castFunPtrToPtr :: FunPtr a -> Ptr b`

`castPtrToFunPtr :: Ptr a -> FunPtr b`

These two functions cast `Ptr`s to `FunPtr`s and vice versa.

Warning: These functions are only valid on architectures that use the same representation for data and function pointers. It is recommended to avoid using these functions; more precisely, only where bindings to external libraries are made whose interface already relies on this assumption, should the use of `castFunPtrToPtr` and `castPtrToFunPtr` be considered at all.

5.29.3. 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  *HsPtr;            /* C representation of a 'Ptr a'    */
typedef void (*HsFunPtr)(void);  /* C representation of a 'FunPtr a' */
```

5.30. ShowFunctions

The `ShowFunctions` module contains the following instance of `Show` for function types

```
instance Show (a -> b) where
showsPrec _ _ = showString "<function>"
```

5.31. ST

This library provides support for *strict* state threads, as described in the PLDI '94 paper by John Launchbury and Simon Peyton Jones [LazyStateThreads]. In addition to the monad `ST`, it also provides mutable variables `STRef` and mutable arrays `STArray`.

```
module ST( module ST, module Monad ) where
import Monad
```

```
data ST s a        - abstract type
runST              :: forall a. (forall s. ST s a) -> a
fixST              :: (a -> ST s a) -> ST s a
unsafeInterleaveST :: ST s a -> ST s a
instance Functor (ST s)
instance Monad   (ST s)

data STRef s a     - mutable variables in state thread s
                   - containing values of type a.
newSTRef           :: a -> ST s (STRef s a)
readSTRef          :: STRef s a -> ST s a
writeSTRef         :: STRef s a -> a -> ST s ()
modifySTRef        :: STRef s a -> (a -> a) -> ST s ()

- deprecated, use modifySTRef
updateSTRef        :: STRef s a -> (a -> a) -> ST s ()
instance Eq (STRef s a)

data STArray s ix elt - mutable arrays in state thread s
                      - indexed by values of type ix
                      - containing values of type a.
newSTArray          :: Ix ix => (ix,ix) -> elt -> ST s (STArray s ix elt)
boundsSTArray       :: Ix ix => STArray s ix elt -> (ix, ix)
readSTArray         :: Ix ix => STArray s ix elt -> ix -> ST s elt
writeSTArray        :: Ix ix => STArray s ix elt -> ix -> elt -> ST s ()
thawSTArray         :: Ix ix => Array ix elt -> ST s (STArray s ix elt)
freezeSTArray       :: Ix ix => STArray s ix elt -> ST s (Array ix elt)
unsafeFreezeSTArray :: Ix ix => STArray s ix elt -> ST s (Array ix elt)
instance Eq (STArray s ix elt)

unsafeIOToST     :: IO   a -> ST s a
stToIO              :: ST s a -> IO a
```

Notes:

- GHC also supports ByteArrays — these aren’t supported by Hugs yet.
- The operations `freezeSTArray` and `thawSTArray` convert mutable arrays to and from immutable arrays. Semantically, they are identical to copying the array and they are usually implemented that way. The operation `unsafeFreezeSTArray` is a faster version of `freezeSTArray` which omits the copying step. It’s a safe substitute for `freezeSTArray` if you don’t modify the mutable array after freezing it.
- Hugs provides `thenLazyST` and `thenStrictST` so that you can import `LazyST` (say) and still use the strict instance in those places where it matters. GHC implements LazyST and ST using different types, so this isn’t possible.

- Operations for coercing an `ST` action into an `IO` one, and vice versa are also provided. Notice that coercing an `IO` action into an `ST` action is 'lossy', since any exception raised within the `IO` action will not be caught within the `ST` monad, as it doesn't support (monadic) exceptions.

5.32. Stable

This module is deprecated, use either `StablePtr` (see Section 5.34) or `StableName` (see Section 5.33).

5.33. StableName

A haskell object can be given a *stable name* by calling `makeStableName` on it. Stable names solve the following problem: suppose you want to build a hash table with Haskell objects as keys, but you want to use pointer equality for comparison; maybe because the keys are large and hashing would be slow, or perhaps because the keys are infinite in size. We can't build a hash table using the address of the object as the key, because objects get moved around by the garbage collector, meaning a re-hash would be necessary after every garbage collection.

Enter stable names. A stable name is an abstract entity that supports equality and hashing, with the following interface:

```
data StableName a - abstract, instance Eq.
makeStableName :: a -> IO (StableName a)
hashStableName :: StableName a -> Int
```

All these operations run in constant time.

Stable names have the following properties:

1. If `sn1 :: StablePtr` and `sn2 :: StablePtr` and `sn1 == sn2` then `sn1` and `sn2` are either the same stable name, or they were created by calls to `makeStableName` on the same object.
2. The reverse is not necessarily true: if two stable names are not equal, it doesn't mean that they don't refer to the same Haskell object (although they probably don't).
3. There is no `freeStableName` operation. Stable names are reclaimed by the runtime system when they are no longer needed.
4. There is no `deRefStableName` operation. You can't get back from a stable name to the original Haskell object. The reason for this is that the existence of a stable name for an object doesn't guarantee the existence of the object itself; it can still be garbage collected.

5. There is a `hashStableName` operation, which converts a stable name to an `Int`. The `Int` returned is not necessarily unique (that is, it doesn't satisfy property (1) above), but it can be used for building hash tables of stable names.

Properties (1) and (2) are similar to stable pointers, but the key differences are that you can't get back to the original object from a stable name, and you can convert one to an `Int` for hashing.

5.34. `StablePtr`

This module is part of the Foreign Function Interface (FFI) and will usually be imported via the module `Foreign` (see Section 5.13). A *stable pointer* is a reference to a Haskell expression that is guaranteed not to be affected by garbage collection, i.e., it will neither be deallocated nor will the value of the stable pointer itself change during garbage collection (ordinary references may be relocated during garbage collection). Consequently, stable pointers can be passed to foreign code, which can handle it as an opaque reference to a Haskell value.

5.34.1. The Standard Interface

```
module StablePtr where

data StablePtr a   - abstract stable reference to a Haskell value
instance Eq StablePtr

newStablePtr       :: a           -> IO (StablePtr a)
deRefStablePtr     :: StablePtr a -> IO a
freeStablePtr      :: StablePtr a -> IO ()
castStablePtrToPtr :: StablePtr a -> Ptr ()
castPtrToStablePtr :: Ptr ()      -> StablePtr a
```

The behaviour of the functions is as follows:

`newStablePtr :: a -> IO (StablePtr a)`

Creates a stable pointer referring to the given Hasell value.

`deRefStablePtr :: StablePtr a -> IO a`

Obtains the Haskell value referenced by a stable pointer, i.e., the same value that was passed to the corresponding call to `makeStablePtr`.

`freeStablePtr :: StablePtr a -> IO ()`

Dissolve the association between the stable pointer and the Haskell value. Afterwards, if the stable pointer is passed to `deRefStablePtr` or `freeStablePtr`, the behaviour is undefined. However, the stable pointer may still be passed to `castStablePtrToPtr`, but the `Ptr ()`

value returned by `castStablePtrToPtr`, in this case, is undefined (in particular, it may be `Ptr.nullPtr`). Nevertheless, the call is guaranteed not to diverge.

`castStablePtrToPtr :: StablePtr a -> Ptr ()`

Coerces a stable pointer to an address. No guarantees are made about the resulting value, except that the original stable pointer can be recovered by `castPtrToStablePtr`. In particular, the address may not refer to a valid memory address and any attempt to pass it to the member functions of the class `Storable` (Section 5.35) will most likely lead to disaster.

`castPtrToStablePtr :: Ptr () -> StablePtr a`

The inverse of `castStablePtrToPtr`, i.e., we have the identity

```
sp == castPtrToStablePtr (castStablePtrToPtr sp)
```

for any stable pointer `sp` on which `freeStablePtr` has *not* been executed yet.

Care must be taken to free stable pointers that are no longer required using the function `freeStablePtr`; otherwise, two bad things can happen:

- The object referenced by the stable pointer will be retained in the heap.
- The runtime system's internal stable pointer table will grow, which imposes an overhead on garbage collection.

5.34.2. 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 *HsStablePtr;  /* C representation of a StablePtr */
```

Note that no assumptions may be made about the values representing stable pointer. In fact, they need not even be valid memory addresses. The only guarantee provided is that if they are passed back to Haskell land, the function `deRefStablePtr` will be able to reconstruct the Haskell value refereed to by the stable pointer.

5.34.3. Deprecated Functions

The following functions are deprecated in the new FFI and the assertions regarding the equality of stable pointers are *not* guaranteed. Do *not* use the following functions if you are interested in portability. Most of these functions are here for legacy reasons and may just vanish one day. You have been warned.

Notes:

- If `sp1 :: StablePtr` and `sp2 :: StablePtr` and `sp1 == sp2` then `sp1` and `sp2` are either the same stable pointer, or they were created by calls to `makeStablePtr` on the same object. Another way to say this is "every time you call `makeStablePtr` on an object you get back the same stable pointer".
- The reverse is not necessarily true: if two stable pointers are not equal, it doesn't mean that they don't refer to the same Haskell object (although they probably don't).

The C interface (which is brought into scope by `#include <Stable.h>`) is as follows:

```
typedef StablePtr /* abstract, probably an unsigned long */
extern StgPtr         deRefStablePtr(StgStablePtr stable_ptr);
static void           freeStablePtr(StgStablePtr sp);
static StgStablePtr   splitStablePtr(StgStablePtr sp);
```

The functions `deRefStablePtr` and `freeStablePtr` are equivalent to the Haskell functions of the same name above.

The function `splitStablePtr` allows a stable pointer to be duplicated without making a new one with `makeStablePtr`. The stable pointer won't be removed from the runtime system's internal table until `freeStablePtr` is called on both pointers.

5.35. `Storable`

The module `Storable` provides most elementary support for marshalling and is part of the language-independent portion of the Foreign Function Interface (FFI) - see also Section 5.13. It provides a class `Storable` and instances of this class for all primitive types that can be stored in raw memory. The member functions of this class facilitate writing values of primitive types to raw memory (which may have been allocated with the above mentioned routines) and reading values from blocks of raw memory. The class, furthermore, includes support for computing the storage requirements and alignment restrictions of storable types.

Memory addresses are represented as values of type `Ptr a`, for some `a` which is an instance of class `Storable`. The type argument to `Ptr` helps provide some valuable type safety in FFI code (you can't mix pointers of different types without an explicit cast), while helping the Haskell type system figure out which marshalling method is needed for a given pointer.

All marshalling between Haskell and a foreign language ultimately boils down to translating Haskell data structures into the binary representation of a corresponding data structure of the foreign language and vice versa. To code this marshalling in Haskell, it is necessary to manipulate primtive data types stored in unstructured memory blocks. The class `Storable` facilitates this manipulation on all types for which it is instantiated, which are the standard basic types of Haskell, the fixed size `IntXX` (Section 5.18) and `WordXX` (Section 5.39) types, stable pointers (Section 5.34), all types from `CTypes` (Section 5.7) and `CTypesISO` (Section 5.8), as well as `Ptrs` (Section 5.29)).

```
class Storable a where
  sizeOf      :: a -> Int
  alignment   :: a -> Int

  peekElemOff :: Ptr a -> Int      -> IO a
  pokeElemOff :: Ptr a -> Int -> a -> IO ()

  peekByteOff :: Ptr a -> Int      -> IO a
  pokeByteOff :: Ptr a -> Int -> a -> IO ()

  peek        :: Ptr a             -> IO a
  poke        :: Ptr a        -> a -> IO ()

  destruct    :: Ptr a             -> IO ()
```

Minimal complete definition: `sizeOf`, `alignment`, one of `peek`, `peekElemOff` or `peekByteOff`, and one of `poke`, `pokeElemOff` and `pokeByteOff`. The behaviour of the member functions is as follows:

`sizeOf :: a -> Int`
`alignment :: a -> Int`

> The function `sizeOf` computes the storage requirements (in bytes) of the argument, and `alignment` computes the alignment constraint of the argument. An alignment constraint `x` is fulfilled by any address divisible by `x`. Both functions *do not evaluate* their argument, but compute the result on the basis of the type of the argument alone.

`peekElemOff :: Ptr a -> Int -> IO a`

> Read a value from a memory area regarded as an array of values of the same kind. The first argument specifies the start address of the array and the second the index into the array (the first element of the array has index `0`). The following equality holds,

```
peekElemOff addr idx = IOExts.fixIO $ \result ->
  peek (addr `plusPtr` (idx * sizeOf result))
```

> Note that this is only a specification, but not necessarily the concrete implementation of the function.

`pokeElemOff :: Ptr a -> Int -> a -> IO ()`

> Write a value to a memory area regarded as an array of values of the same kind. The following equality holds,

```
pokeElemOff addr idx x =
  poke (addr `plusPtr` (idx * sizeOf x)) x
```

`peekByteOff :: Ptr a -> Int -> IO a`

Read a value from a memory location given by a base address and offset. The following equality holds,

```
peekByteOff addr off = peek (addr `plusPtr` off)
```

`pokeByteOff :: Ptr a -> Int -> a -> IO ()`

Write a value to a memory location given by a base address and offset. The following equality holds,

```
pokeByteOff addr off x = poke (addr `plusPtr` off) x
```

`peek :: Ptr a -> IO a`

Read a value from the given memory location.

`poke :: Ptr a -> a -> IO ()`

Write the given value to the given memory location.

`destruct :: Ptr a -> IO ()`

Perform finalization (e.g. free an additional memory associated with the object) if `poke` does something which needs to be undone. A call to `poke` followed by `destruct` should leave resources in the same state as before. The default definition of `destruct` does nothing, as most types don't need any finalization.

Note that the peek and poke functions might require properly aligned addresses to function correctly. This is architecture dependent; thus, portable code should ensure that when peeking or poking values of some type `a`, the alignment constraint for `a`, as given by the function `alignment` is fulfilled.

5.36. StorableArray

A storable array is a mutable array living in the IO monad, which stores its contents in a contiguous memory block living in the C heap. Elements are stored according to the class `Storable` (Section 5.35). You can obtain the pointer to the array contents to manipulate elements from languages like C.

It's similar to `IOUArray` but slower. Its advantage is that it's compatible with C.

The array type constructor is called `StorableArray`. It must be applied to an index type (of class `Ix`) and element type (of class `Storable`), providing an array type which is an instance of class `MArray` (Section 5.21).

The pointer to the array contents is obtained by `withStorableArray`. The idea is similar to `ForeignPtr` (used internally here by the way). The pointer should be used only during execution of the IO action retured by the function passed as argument to `withStorableArray`:

```
withStorableArray :: StorableArray i e -> (Ptr e -> IO a) -> IO a
```

If you want to use it afterwards, ensure that you `touchStorableArray` after the last use of the pointer, so the array is not freed too early:

```
touchStorableArray :: StorableArray i e -> IO ()
```

5.37. SystemExts

The `SystemExts` module contains functionality that goes beyond what the Haskell 98 module `System` provides. That is, functionality that provides access to the underlying OS' facilities in an OS-indepedent manner.

Notice that `SystemExts` shares the goal of `System`. That is, it aims to provide functionality that's supported by all platforms. So, if you're looking to do serious system programming for a particular (family) of platforms, you really want to check out the libraries provided for the platform in question as well. e.g., The `Posix` library for `POSIX.1`-conforming platforms, the `Win32` library for Win32 platforms.

`SystemExts` exports the following:

```
rawSystem     :: String -> IO ExitCode
withArgs      :: [String] -> IO a -> IO a
withProgName  :: String -> IO a -> IO a
```

Notes:

- `rawSystem` provides the exact same behaviour as `System.system`, except that the system command isn't invoked via an shell / command interpreter.

  Not involving your platform's shell / command interpreter is quicker if you don't need its functionality, and it avoids running into limitations imposed by the shell / command interpreter. For instance, Win32 command interpreters place a limit on the length of the command they can execute (~4k), which sometimes gets in the way of what you want to do.

- The `withArgs` action lets you change the value returned by `System.getArgs` while executing an IO action.

  When the action has finished executing (or if it raises an exception), the argument vector of `System.getArgs` is restored.

- The `withProgName` action lets you change the program name string returned by `System.getProgName` while executing an IO action.

  As `withArgs`, when the action has finished executing (or if it raises an exception), the program name string `System.getArgs` is restored.

5.38. `Weak`

The `Weak` library provides a "weak pointer" abstraction, giving the user some control over the garbage collection of specified objects, and allowing objects to be "finalized" with an arbitrary Haskell IO computation when they die.

Weak pointers partially replace the old foreign object interface, as we will explain later.

5.38.1. Module Signature

```
module Weak (
Weak,      - abstract
- instance Eq (Weak v)

mkWeak,       - :: k -> v -> Maybe (IO ()) -> IO (Weak v)
deRefWeak,  - :: Weak v -> IO (Maybe v)
finalize, - :: Weak v -> IO ()

- Not yet implemented
- replaceFinalizer - :: Weak v -> IO () -> IO ()

mkWeakPtr,  - :: k -> Maybe (IO ()) -> IO (Weak k)
mkWeakPair,  - :: k -> v -> Maybe (IO ()) -> IO (Weak (k,v))
addFinalizer  - :: k -> IO () -> IO ()
   ) where
```

5.38.2. Weak pointers

In general terms, a weak pointer is a reference to an object that is not followed by the garbage collector — that is, the existence of a weak pointer to an object has no effect on the lifetime of that object. A weak pointer can be de-referenced to find out whether the object it refers to is still alive or not, and if so to return the object itself.

Weak pointers are particularly useful for caches and memo tables. To build a memo table, you build a data structure mapping from the function argument (the key) to its result (the value). When you apply the function to a new argument you first check whether the key/value pair is already in the memo table. The key point is that the memo table itself should not keep the key and value alive. So the table should contain a weak pointer to the key, not an ordinary pointer. The pointer to the value must not be weak, because the only reference to the value might indeed be from the memo table.

So it looks as if the memo table will keep all its values alive for ever. One way to solve this is to purge the table occasionally, by deleting entries whose keys have died.

The weak pointers in this library support another approach, called *finalization*. When the key referred to by a weak pointer dies, the storage manager arranges to run a programmer-specified finalizer. In the case of memo tables, for example, the finalizer could remove the key/value pair from the memo table.

Another difficulty with the memo table is that the value of a key/value pair might itself contain a pointer to the key. So the memo table keeps the value alive, which keeps the key alive, even though there may be no other references to the key so both should die. The weak pointers in this library provide a slight generalisation of the basic weak-pointer idea, in which each weak pointer actually contains both a key and a value. We describe this in more detail below.

5.38.3. The simple interface

```
mkWeakPtr    :: a -> Maybe (IO ()) -> IO (Weak a)
deRefWeak    :: Weak a -> IO (Maybe a)
addFinalizer :: a -> IO () -> IO ()
```

`mkWeakPtr` takes a value of any type `a`, and maybe a finalizer of type `IO ()`, and returns a weak pointer object referring to the value, of type `Weak a`. It is in the `IO` monad because it has the side effect of arranging that the finalizer (if there is one) will be run when the object dies. In what follows, a "weak pointer object", or "weak pointer" for short, means precisely "a Haskell value of type `Weak t`" for some type `t`. A weak pointer (object) is a first-class Haskell value; it can be passed to functions, stored in data structures, and so on.

`deRefWeak` dereferences a weak pointer, returning `Just v` if the value is still alive. If the key has already died, then `deRefWeak` returns `Nothing`; that's why it's in the `IO` monad—the return value of `deRefWeak` depends on when the garbage collector runs.

`addFinalizer` is just another name for `mkWeakPtr` except that it throws the weak pointer itself away. (The runtime system will remember that the weak pointer and hence the finalizer exists even if the program has forgotten it.)

```
addFinalizer :: a -> IO () -> IO ()
addFinalizer v f = do { mkWeakPtr v f; return () }
```

The effect of `addFinalizer` is simply that the finalizer runs when the referenced object dies.

The following properties hold:

- `deRefWeak` returns the original object until that object is considered dead; it returns `Nothing` subsequently.
- Every finalizer will eventually be run, exactly once, either soon after the object dies, or at the end of the program. There is no requirement for the programmer to hold onto the weak pointer itself; finalization is completely unaffected by whether the weak pointer itself is alive.
- There may be multiple weak pointers to a single object. In this case, the finalizers for each of these weak pointers will all be run in some arbitrary order, or perhaps concurrently, when the object

dies. If the programmer specifies a finalizer that assumes it has the only reference to an object (for example, a file that it wishes to close), then the programmer must ensure that there is only one such finalizer.

- The storage manager attempts to run the finalizer(s) for an object soon after the object dies, but promptness is not guaranteed. (What is guaranteed is that the finalizer will eventually run, exactly once.)
- At the moment when a finalizer is run, a call to `deRefWeak` will return `Nothing`.
- A finalizer may contain a pointer to the object, but that pointer will not keep the object alive. For example:

```
f :: Show a => a -> IO a
f x = addFinalizer x (print (show x))
```

Here the finalizer `print (show x)` contains a reference to `x` itself, but that does not keep `x` alive. When that is the only reference to `x`, the finalizer is run; and the message appears on the screen.

- A finalizer may even resurrect the object, by (say) storing it in some global data structure.
- If there are no other threads to run, the runtime system will check for runnable finalizers before declaring the system to be deadlocked.

5.38.4. The general interface

The `Weak` library offers a slight generalisation of the simple weak pointers described so far:

```
mkWeak :: k -> v -> Maybe (IO ()) -> IO (Weak v)
```

`mkWeak` takes a key of any type `k` and a value of any type `v`, as well as a finalizer, and returns a weak pointer of type `Weak v`.

`deRefWeak` returns the *value* only, not the key, as its type (given above) implies:

```
deRefWeak :: Weak a -> IO (Maybe a)
```

However, `deRefWeak` returns `Nothing` if the *key*, not the value, has died. Furthermore, references from the value to the key do not keep the key alive, in the same way that the finalizer does not keep the key alive.

Simple weak pointers are readily defined in terms of these more general weak pointers:

```
mkWeakPtr :: a -> Maybe (IO ()) -> IO (Weak a)
mkWeakPtr v f = mkWeak v v f
```

These more general weak pointers are enough to implement memo tables properly.

A weak pointer can be finalized early, using the `finalize` operation:

```
finalize :: Weak v -> IO ()
```

5.38.5. A precise semantics

The above informal specification is fine for simple situations, but matters can get complicated. In particular, it needs to be clear exactly when a key dies, so that any weak pointers that refer to it can be finalized. Suppose, for example, the value of one weak pointer refers to the key of another...does that keep the key alive?

The behaviour is simply this:

- If a weak pointer (object) refers to an *unreachable* key, it may be finalized.
- Finalization means (a) arrange that subsequent calls to `deRefWeak` return `Nothing`; and (b) run the finalizer.

This behaviour depends on what it means for a key to be reachable. Informally, something is reachable if it can be reached by following ordinary pointers from the root set, but not following weak pointers. We define reachability more precisely as follows A heap object is reachable if:

- It is a member of the *root set*.
- It is directly pointed to by a reachable object, other than a weak pointer object.
- It is a weak pointer object whose key is reachable.
- It is the value or finalizer of an object whose key is reachable.

The root set consists of all runnable threads, and all stable pointers (see Section 5.34). NOTE: currently all top-level objects are considered to be reachable, although we hope to remove this restriction in the future. A `Char` or small `Int` will also be constantly reachable, since the garbage collector replaces heap-resident `Char`s and small `Int`s with pointers to static copies.

Notice that a pointer to the key from its associated value or finalizer does not make the key reachable. However, if the key is reachable some other way, then the value and the finalizer are reachable, and so, therefore, are any other keys they refer to directly or indirectly.

5.38.6. Finalization for foreign objects

A foreign object is some data that lives outside the Haskell heap, for example some `malloc`ed data in C land. It's useful to be able to know when the Haskell program no longer needs the `malloc`ed data, so it can be `free`d. We can use weak pointers and finalizers for this, but we have to be careful: the foreign data is usually referenced by an address, ie. an `Addr` (see Section 5.1), and we must retain the invariant that *if the Haskell program still needs the foreign object, then it retains the `Addr` object in the heap*. This invariant isn't guaranteed to hold if we use `Addr`, because an `Addr` consists of a box around a raw address `Addr#`. If the Haskell program can manipulate the `Addr#` object independently of the heap-resident `Addr`, then the foreign object could be inadvertently finalized early, because a weak pointer to the `Addr` would find no more references to its key and trigger the finalizer despite the fact that the program still holds the `Addr#` and intends to use it.

To avoid this somewhat subtle race condition, we use another type of foreign address, called `ForeignObj` (see Section 5.13). Historical note: `ForeignObj` is identical to the old `ForeignObj` except that it no longer supports finalization - that's provided by the weak pointer/finalization mechanism above.

A `ForeignObj` is basically an address, but the `ForeignObj` itself is a heap-resident object and can therefore be watched by weak pointers. A `ForeignObj` can be passed to C functions (in which case the C function gets a straightforward pointer), but it cannot be decomposed into an `Addr#`.

5.39. `Word`

This library provides unsigned integers of various sizes. The types supported are as follows:

type	number of bits
Word8	8
Word16	16
Word32	32
Word64	64

For each type *W* above, we provide the following functions and instances. The type *I* refers to the signed integer type of the same size.

```
data W            - Unsigned Ints
instance Eq       W
instance Ord      W
instance Show     W
instance Read     W
instance Bounded  W
instance Num      W
instance Real     W
instance Integral W
instance Enum     W
instance Ix       W
instance Bits     W
```

5.39.1. Notes

- All arithmetic is performed modulo 2^n. One non-obvious consequence of this is that `negate` should *not* raise an error on negative arguments.

- For coercing between any two integer types, use `fromIntegral`, which is specialized for all the common cases so should be fast enough. Coercing word types to and from integer types preserves representation, not sign.
- It would be very natural to add a type a type `Natural` providing an unbounded size unsigned integer—just as `Integer` provides unbounded size signed integers. We do not do that yet since there is no demand for it. Doing so would require `Bits.bitSize` to return `Maybe Int`.
- The rules that hold for `Enum` instances over a bounded type such as `Int` (see the section of the Haskell report dealing with arithmetic sequences) also hold for the `Enum` instances over the various `Word` types defined here.
- Right and left shifts by amounts greater than or equal to the width of the type result in a zero result. This is contrary to the behaviour in C, which is undefined; a common interpretation is to truncate the shift count to the width of the type, for example `1 « 32 == 1` in some C implementations.

5.39.2. Implementation notes

- Hugs only provides `Eq`, `Ord`, `Read` and `Show` instances for `Word64` at the moment.

Notes

1. GHC currently does not throw `ArithException`s.
2. GHC currently does not throw `ArrayException`s.
3. Which thread receives this exception is currently undefined.
4. GHC currently does not throw `HeapOverflow` exceptions.
5. The default implementation of `unsafeFreeze` is `freeze`, but it is expected that specialised versions which omit the copy are provided for the common array types.

Chapter 6. The `net` package: networking support

(Darren Moffat supplied the initial version of this library.)

6.1. `BSD`: System database info

The `BSD` module provides functions to get at system-database info; pretty straightforward if you're into this sort of thing:

```
type HostName          = String

getHostName           :: IO Hostname

data ServiceEntry = ServiceEntry  {
   serviceName        :: ServiceName,   - Official Name
   serviceAliases     :: [ServiceName], - aliases
   servicePort        :: PortNumber,    - Port Number (network byte order)
   serviceProtocol    :: ProtocolName   - Protocol
 }
type ServiceName       = String

getServiceByName      :: ServiceName -> ProtocolName -> IO ServiceEntry
getServiceByPort      :: PortNumber  -> ProtocolName -> IO ServiceEntry
getServicePortNumber :: ServiceName -> IO PortNumber

- not available con Cygwin/Mingw
getServiceEntry       :: IO ServiceEntry
setServiceEntry       :: Bool -> IO ()
endServiceEntry       :: IO ()
getServiceEntries     :: Bool -> IO [ServiceEntry]

type ProtocolName      = String
type ProtocolNumber    = Int   - from SocketPrim

data ProtocolEntry = ProtocolEntry  {
     protoName        :: ProtocolName,   - Official Name
     protoAliases     :: [ProtocolName], - aliases
     protoNumber      :: ProtocolNumber  - Protocol Number
 }

getProtocolByName     :: ProtocolName   -> IO ProtocolEntry
getProtocolByNumber   :: ProtocolNumber -> IO ProtcolEntry
getProtocolNumber     :: ProtocolName   -> ProtocolNumber
```

```
 - not available con Cygwin/Mingw
setProtocolEntry     :: Bool -> IO ()
getProtocolEntry     :: IO ProtocolEntry
endProtocolEntry     :: IO ()
getProtocolEntries   :: Bool -> IO [ProtocolEntry]

- from SocketPrim
newtype PortNumber    = -
instance of Eq, Ord, Enum, Num, Real, Integral, Show
   PNum Int             - 16-bit value stored in network byte order.
mkPortNumber         :: Int -> PortNumber

data HostEntry = HostEntry {
     hostName        :: HostName,     - Official Name
     hostAliases     :: [HostName],   - aliases
     hostFamily      :: Family,       - Host Type (currently AF_INET)
     hostAddresses   :: [HostAddress] -
Set of Network Addresses  (in network byte order)
 }

getHostByName        :: HostName -> IO HostEntry
getHostByAddr        :: HostAddress -> Family -> IO HostEntry
hostAddress          :: HostEntry -> HostAddress

- not available con Cygwin/Mingw
setHostEntry         :: Bool -> IO ()
getHostEntry         :: IO HostEntry
endHostEntry         :: IO ()
getHostEntries       :: Bool -> IO [HostEntry]

type NetworkAddr      = Word - host byte order
type NetworkName      = String

data NetworkEntry =
  NetworkEntry {
     networkName     :: NetworkName,   - official name
     networkAliases  :: [NetworkName], - aliases
     networkFamily   :: Family,        - type
     networkAddress  :: NetworkAddr
   }

- not available con Cygwin/Mingw
getNetworkByName     :: NetworkName -> IO NetworkEntry
getNetworkByAddr     :: NetworkAddr -> Family -> IO NetworkEntry
setNetworkEntry      :: Bool -> IO ()
getNetworkEntry      :: IO NetworkEntry
endNetworkEntry      :: IO ()
getNetworkEntries    :: Bool -> IO [NetworkEntry]
```

```
- if available
symlink              :: String -> String -> IO ()
readlink             :: String -> IO String
```

6.2. Socket: The high-level networking interface

The `Socket` interface is a "higher-level" interface to sockets, and it is what we recommend. Please tell us if the facilities it offers are inadequate to your task! The interface is relatively modest:

```
data Socket     - instance of: Eq, Show

data PortID =
      Service String        - Service Name eg "ftp"
    | PortNumber PortNumber - User defined Port Number
    | UnixSocket String     - Unix family socket in file system,
                            - not available con Cygwin/Mingw
type Hostname    = String

connectTo       :: Hostname -> PortID -> IO Handle
listenOn        :: PortID -> IO Socket

accept :: Socket -> IO (Handle, HostName, PortNumber)

sendTo          :: Hostname -> PortID -> String -> IO ()
recvFrom        :: Hostname -> PortID -> IO String

socketPort      :: Socket -> IO PortID

withSocketsDo   :: IO a -> IO a

data PortNumber - instance of Eq, Ord, Enum, Num, Real, Integral, Show
mkPortNumber    :: Int -> PortNumber
```

6.2.1. Buffering

The `Handle` returned by `connectTo` and `accept` is unbuffered by default. To improve performance when shifting large amounts of data, it is wise to enable block buffering on the `Handle` with `IO.hSetBuffering`. For really fast I/O, it might also be worth looking at the `hGetBuf` and `hPutBuf` family of functions in `IOExts` (see Section 5.19).

6.2.2. SIGPIPE

On Unix, when reading from a socket and the writing end is closed by the remote client, the program is normally sent a SIGPIPE signal by the operating system. The default behaviour when a SIGPIPE is received is to terminate the program silently, which can be somewhat confusing if you haven't encountered this before. The solution is to specify that SIGPIPE is to be ignored, using the POSIX library (Section 8.2):

6.2.3. Sockets on Windows

On Windows, you need to initialise WinSock before trying to use sockets, i.e.

```
main = withSocketDo $ do {...}
```

6.3. SocketPrim: The low-level socket binding

The SocketPrim module is for when you want full control over the sockets, exposing the C socket API. Your best bet for documentation is to look at the code—really!— normally in fptools/hslibs/net/SocketPrim.lhs.

```
data Socket             - instance of: Eq, Show

- your mileage may vary depending on the OS you use...
data Family = instance of: Eq, Ord, Ix, Show
     AF_802             - IEEE 802.2, also ISO 8802
   | AF_APPLETALK       - Apple Talk
   | AF_ARP             - (rev.) addr. res. prot. (RFC 826)
   | AF_AX25
   | AF_CCITT           - CCITT protocols, X.25 etc
   | AF_CHAOS           - mit CHAOS protocols
   | AF_CNT             - Computer Network Technology
   | AF_COIP            - connection-oriented IP, aka ST II
   | AF_CTF             - Common Trace Facility
   | AF_DATAKIT         - datakit protocols
   | AF_DECnet          - DECnet
   | AF_DLI             - DEC Direct data link interface
   | AF_ECMA            - european computer manufacturers
   | AF_GOSSIP          - US Government OSI
   | AF_HYLINK          - NSC Hyperchannel
   | AF_IMPLINK         - arpanet imp addresses
   | AF_INET            - internetwork: UDP, TCP, etc
   | AF_INET6           - IPv6
   | AF_INTF            - Debugging use only
   | AF_IPX             - Novell Internet Protocol
   | AF_ISDN            - Integrated Services Digital Network
   | AF_ISO             - ISO protocols
```

```
    | AF_LAT             - LAT
    | AF_LINK            - Link layer interface
    | AF_MAX
    | AF_NATM            - native ATM access
    | AF_NBS             - nbs protocols
    | AF_NDD
    | AF_NETBIOS         - NetBios-style addresses
    | AF_NETMAN          - DNA Network Management
    | AF_NETWARE
    | AF_NIT             - Network Interface Tap
    | AF_NS              - XEROX NS protocols
    | AF_OSI             - OSI protocols
    | AF_OSINET          - AFI
    | AF_PUP             - pup protocols: e.g. BSP
    | AF_RAW             - Link layer interface
    | AF_RIF             - raw interface
    | AF_ROUTE           - Internal Routing Protocol
    | AF_SIP             - Simple Internet Protocol
    | AF_SNA             - IBM SNA
    | AF_UNIX            - local to host (pipes, portals
    | AF_UNSPEC          - unspecified
    | AF_WAN             - Wide Area Network protocols
    | AF_X25             - CCITT X.25
    | Pseudo_AF_HDRCMPLT - Used by BPF to not rewrite hdrs in iface output
    | Pseudo_AF_KEY      - Internal key-management function
    | Pseudo_AF_PIP      - Help Identify PIP packets
    | Pseudo_AF_RTIP     - Help Identify RTIP packets
    | Pseudo_AF_XTP      - eXpress Transfer Protocol (no AF)

data Socket
  = MkSocket Int        - File Descriptor
             Family
             SocketType
             Int        - Protocol Number
             (IORef SocketStatus) - Status Flag

data SockAddr
  = SockAddrUnix String - not available con Cygwin/Mingw
  | SockAddrInet PortNumber HostAddress

type HostAddress = Word

data ShutdownCmd = ShutdownReceive | ShutdownSend | ShutdownBoth

type ProtocolNumber = Int

socket          :: Family -> SocketType -> ProtocolNumber -> IO Socket
connect         :: Socket -> SockAddr -> IO ()
bindSocket      :: Socket -> SockAddr -> IO ()
```

```
listen          :: Socket -> Int -> IO ()
accept          :: Socket -> IO (Socket, SockAddr)
getPeerName     :: Socket -> IO SockAddr
getSocketName   :: Socket -> IO SockAddr

socketPort      :: Socket -> IO PortNumber

socketToHandle  :: Socket -> IOMode -> IO Handle

send            :: Socket -> String -> IO Int
recv            :: Socket -> Int -> IO String

sendTo          :: Socket -> String -> SockAddr -> IO Int
recvFrom        :: Socket -> Int -> IO (String, Int, SockAddr)

inet_addr       :: String -> IO HostAddress
inet_ntoa       :: HostAddress -> IO String

sIsConnected    :: Socket -> IO Bool
sIsBound        :: Socket -> IO Bool
sIsListening    :: Socket -> IO Bool
sIsReadable     :: Socket -> IO Bool
sIsWritable     :: Socket -> IO Bool
shutdown        :: Socket -> ShutdownCmd -> IO ()
sClose          :: Socket -> IO ()

data SocketType
    - NOTE: not all of the socket types are supported on every platform.
    = Stream
    | Datagram
    | Raw
    | RDM
    | SeqPacket
    | Packet
    deriving (Eq, Ord, Ix, Show)

data SocketOption =
      Debug          - SO_DEBUG
    | ReuseAddr      - SO_REUSEADDR
    | Type           - SO_TYPE
    | SoError        - SO_ERROR
    | DontRoute      - SO_DONTROUTE
    | Broadcast      - SO_BROADCAST
    | SendBuffer     - SO_SNDBUF
    | RecvBuffer     - SO_RCVBUF
    | KeepAlive      - SO_KEEPALIVE
    | OOBInline      - SO_OOBINLINE
    | MaxSegment     - TCP_MAXSEG, not available con Cygwin/Mingw
    | NoDelay        - TCP_NODELAY
```

```
getSocketOption :: Socket -> SocketOption -> IO Int
setSocketOption :: Socket -> SocketOption -> Int -> IO ()

newtype PortNumber = -
instance of Eq, Ord, Enum, Num, Real, Integral, Show
   PNum Int          - 16-bit value stored in network byte order.

aNY_PORT        :: PortNumber
iNADDR_ANY      :: HostAddress
sOMAXCONN       :: Int
maxListenQueue  :: Int
```

6.4. URI

The URI library provides utilities for parsing and manipulating Uniform Resource Identifiers (a more general form of Uniform Resource Locators, or URLs). URIs are described in RFC 2396 (http://www.faqs.org/rfcs/rfc2396.html).

```
module URI where

data URI {
  scheme, :: String,
  authority, :: String,
  path, :: String,
  query, :: String,
  fragment :: String
  }
instance Show URI

parseURI        :: String -> Maybe URI

relativeTo      :: URI -> URI -> Maybe URI

- support for putting strings into URI-friendly
- escaped format and getting them back again.
- Can't be done transparently, because certain characters
- have different meanings in different kinds of URI.

reserved  :: Char -> Bool
unreserved  :: Char -> Bool
isAllowedInURI :: Char -> Bool
escapeString    :: String -> (Char->Bool) -> String
unEscapeString :: String -> String
```

Chapter 7. The `num` package: numeric operations

This category is currently empty.

Chapter 8. The `posix` package: POSIX support

The `Posix` interface gives you access to the set of OS services standardised by POSIX 1003.1b (or the *IEEE Portable Operating System Interface for Computing Environments* - IEEE Std. 1003.1). The interface is accessed by `import Posix` and adding `-package posix` on your command-line.

The Posix package is *not* supported under Windows. We've looked into various ways of providing support, and other than using Cygwin, none is particularly attractive. If you want Posix support under Windows, try building GHC for Cygwin; we don't currently do this, but it is mostly supported.

8.1. Posix data types

```
data ByteCount  - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `ByteCount` is a primitive of type `unsigned`. At a minimum, an conforming implementation must support values in the range `[0, UINT_MAX]`.

```
data ClockTick  - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `ClockTick` is a primitive of type `clock_t`, which is used to measure intervals of time in fractions of a second. The resolution is determined by `getSysVar ClockTick`.

```
data DeviceID  - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `DeviceID` is a primitive of type `dev_t`. It must be an arithmetic type.

```
data EpochTime - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `EpochTime` is a primitive of type `time_t`, which is used to measure seconds since the Epoch. At a minimum, the implementation must support values in the range `[0, INT_MAX]`.

```
data FileID - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `FileID` is a primitive of type `ino_t`. It must be an arithmetic type.

```
data FileMode - instance of : Eq
```

A `FileMode` is a primitive of type `mode_t`. It must be an arithmetic type.

```
data FileOffset - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `FileOffset` is a primitive of type `off_t`. It must be an arithmetic type.

```
data GroupID - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `GroupID` is a primitive of type `gid_t`. It must be an arithmetic type.

```
data Limit - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `Limit` is a primitive of type `long`. At a minimum, the implementation must support values in the range `[LONG_MIN, LONG_MAX]`.

```
data LinkCount - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `LinkCount` is a primitive of type `nlink_t`. It must be an arithmetic type.

```
data ProcessID - instances of : Eq Ord Num Real Integral Ix Enum Show
type ProcessGroupID = ProcessID
```

A `ProcessID` is a primitive of type `pid_t`. It must be a signed arithmetic type.

```
data UserID - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `UserID` is a primitive of type `uid_t`. It must be an arithmetic type.

```
data DirStream
```

A `DirStream` is a primitive of type `DIR *`.

```
data FileStatus
```

A `FileStatus` is a primitive of type `struct stat`.

```
data GroupEntry
```

A `GroupEntry` is a primitive of type `struct group`.

```
data ProcessTimes
```

`ProcessTimes` is a primitive structure containing a `clock_t` and a `struct tms`.

```
data SignalSet
```

An `SignalSet` is a primitive of type `sigset_t`.

```
data SystemID
```

A `SystemID` is a primitive of type `struct utsname`.

```
data TerminalAttributes
```

`TerminalAttributes` is a primitive of type `struct termios`.

```
data UserEntry
```

A `UserEntry` is a primitive of type `struct passwd`.

```
data BaudRate = B0 | B50 | B75 | B110 | B134 | B150 | B200 | B300 | B600
              | B1200 | B1800 | B2400 | B4800 | B9600 | B19200 | B38400
              deriving (Eq, Show)

data Fd

intToFd :: Int -> Fd - use with care.
fdToInt :: Fd -> Int     - ditto.

data FdOption = AppendOnWrite
              | CloseOnExec
              | NonBlockingRead

data ControlCharacter = EndOfFile
                      | EndOfLine
                      | Erase
                      | Interrupt
                      | Kill
                      | Quit
                      | Suspend
                      | Start
                      | Stop

type ErrorCode = Int

type FileLock = (LockRequest, SeekMode, FileOffset, FileOffset)
-                             whence    start       length

data FlowAction = SuspendOutput | RestartOutput | Transmit-
Stop | TransmitStart

data Handler = Default | Ignore | Catch (IO ())

data LockRequest = ReadLock | WriteLock | Unlock
                 deriving (Eq, Show)
```

```
data OpenMode = ReadOnly | WriteOnly | ReadWrite

data PathVar = LinkLimit
             | InputLineLimit
             | InputQueueLimit
             | FileNameLimit
             | PathNameLimit
             | PipeBufferLimit
             | SetOwnerAndGroupIsRestricted
             | FileNamesAreNotTruncated

data QueueSelector = InputQueue | OutputQueue | BothQueues

type Signal = Int

data SysVar = ArgumentLimit
            | ChildLimit
            | ClockTick
            | GroupLimit
            | OpenFileLimit
            | PosixVersion
            | HasSavedIDs
            | HasJobControl

data TerminalMode = InterruptOnBreak       - BRKINT
                  | MapCRtoLF              - ICRNL
                  | IgnoreBreak            - IGNBRK
                  | IgnoreCR               - IGNCR
                  | IgnoreParityErrors     - IGNPAR
                  | MapLFtoCR              - INLCR
                  | CheckParity            - INPCK
                  | StripHighBit           - ISTRIP
                  | StartStopInput         - IXOFF
                  | StartStopOutput        - IXON
                  | MarkParityErrors       - PARMRK
                  | ProcessOutput          - OPOST
                  | LocalMode              - CLOCAL
                  | ReadEnable             - CREAD
                  | TwoStopBits            - CSTOPB
                  | HangupOnClose          - HUPCL
                  | EnableParity           - PARENB
                  | OddParity              - PARODD
                  | EnableEcho             - ECHO
                  | EchoErase              - ECHOE
                  | EchoKill               - ECHOK
                  | EchoLF                 - ECHONL
                  | ProcessInput           - ICANON
                  | ExtendedFunctions      - IEXTEN
```

```
                | KeyboardInterrupts       - ISIG
                | NoFlushOnInterrupt       - NOFLSH
                | BackgroundWriteInterrupt - TOSTOP

data TerminalState = Immediately | WhenDrained | WhenFlushed

data ProcessStatus = Exited ExitCode
                   | Terminated Signal
                   | Stopped Signal
                   deriving (Eq, Show)
```

8.2. Posix Process Primitives

```
forkProcess :: IO (Maybe ProcessID)
```

`forkProcess` calls `fork`, returning `Just pid` to the parent, where `pid` is the ProcessID of the child, and returning `Nothing` to the child.

```
executeFile :: FilePath                     - Command
            -> Bool                         - Search PATH?
            -> [String]                     - Arguments
            -> Maybe [(String, String)]     - Environment
            -> IO ()
```

`executeFile cmd args env` calls one of the `execv*` family, depending on whether or not the current PATH is to be searched for the command, and whether or not an environment is provided to supersede the process's current environment. The basename (leading directory names suppressed) of the command is passed to `execv*` as `arg[0]`; the argument list passed to `executeFile` therefore begins with `arg[1]`.

```
Search PATH?    Supersede environ?      Call
~~~~~~~~~~~~    ~~~~~~~~~~~~~~~~~~      ~~~~~~~
False           False                   execv
False           True                    execve
True            False                   execvp
True            True                    execvpe*
```

Note that `execvpe` is not provided by the POSIX standard, and must be written by hand. Care must be taken to ensure that the search path is extracted from the original environment, and not from the environment to be passed on to the new image.

NOTE: In general, sharing open files between parent and child processes is potential bug farm, and should be avoided unless you really depend on this 'feature' of POSIX' `fork()` semantics. Using

Haskell, there's the extra complication that arguments to `executeFile` might come from files that are read lazily (using `hGetContents`, or some such.) If this is the case, then for your own sanity, please ensure that the arguments to `executeFile` have been fully evaluated before calling `forkProcess` (followed by `executeFile`.) Consider yourself warned :-)

A successful `executeFile` overlays the current process image with a new one, so it only returns on failure.

```
runProcess :: FilePath                      - Command
           -> [String]                      - Arguments
           -> Maybe [(String, String)]      - Environment (Nothing -
> Inherited)
           -> Maybe FilePath                - Working directory (Nothing -
> inherited)
           -> Maybe Handle                  - stdin  (Nothing -> inherited)
           -> Maybe Handle                  - stdout (Nothing -> inherited)
           -> Maybe Handle                  - stderr (Nothing -> inherited)
           -> IO ()
```

`runProcess` is our candidate for the high-level OS-independent primitive.

`runProcess cmd args env wd inhdl outhdl errhdl` runs **cmd** (searching the current `PATH`) with arguments `args`. If `env` is `Just pairs`, the command is executed with the environment specified by `pairs` of variables and values; otherwise, the command is executed with the current environment. If `wd` is `Just dir`, the command is executed with working directory `dir`; otherwise, the command is executed in the current working directory. If `{in,out,err}`hdl} is `Just handle`, the command is executed with the `Fd` for `std{in,out,err}` attached to the specified `handle`; otherwise, the `Fd` for `std{in,out,err}` is left unchanged.

```
getProcessStatus :: Bool              - Block?
                 -> Bool              - Stopped processes?
                 -> ProcessID
                 -> IO (Maybe ProcessStatus)
```

`getProcessStatus blk stopped pid` calls `waitpid`, returning `Just tc`, the `ProcessStatus` for process `pid` if it is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
getGroupProcessStatus :: Bool         - Block?
                      -> Bool         - Stopped processes?
                      -> ProcessGroupID
                      -> IO (Maybe (ProcessID, ProcessStatus))
```

`getGroupProcessStatus blk stopped pgid` calls `waitpid`, returning `Just (pid, tc)`, the `ProcessID` and `ProcessStatus` for any process in group `pgid` if one is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
getAnyProcessStatus :: Bool           - Block?
                    -> Bool           - Stopped processes?
                    -> IO (Maybe (ProcessID, ProcessStatus))
```

`getAnyProcessStatus blk stopped` calls `waitpid`, returning `Just (pid, tc)`, the `ProcessID` and `ProcessStatus` for any child process if one is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
exitImmediately :: ExitCode -> IO ()
```

`exitImmediately status` calls `_exit` to terminate the process with the indicated exit `status`. The operation never returns.

```
getEnvironment :: IO [(String, String)]
```

`getEnvironment` parses the environment variable mapping provided by `environ`, returning `(variable, value)` pairs. The operation never fails.

```
setEnvironment :: [(String, String)] -> IO ()
```

`setEnvironment` replaces the process environment with the provided mapping of `(variable, value)` pairs.

```
getEnvVar :: String -> IO String
```

`getEnvVar var` returns the value associated with variable `var` in the current environment (identical functionality provided through standard Haskell library function `System.getEnv`).

The operation may fail with:

`NoSuchThing`

The variable has no mapping in the current environment.

```
setEnvVar :: String -> String -> IO ()
```

`setEnvVar var val` sets the value associated with variable `var` in the current environment to be `val`. Any previous mapping is superseded.

```
removeEnvVar :: String -> IO ()
```

`removeEnvVar var` removes any value associated with variable `var` in the current environment. Deleting a variable for which there is no mapping does not generate an error.

```
nullSignal :: Signal
nullSignal = 0

backgroundRead, sigTTIN        :: Signal
backgroundWrite, sigTTOU       :: Signal
continueProcess, sigCONT       :: Signal
floatingPointException, sigFPE :: Signal
illegalInstruction, sigILL     :: Signal
internalAbort, sigABRT         :: Signal
keyboardSignal, sigINT         :: Signal
keyboardStop, sigTSTP          :: Signal
keyboardTermination, sigQUIT   :: Signal
killProcess, sigKILL           :: Signal
lostConnection, sigHUP         :: Signal
openEndedPipe, sigPIPE         :: Signal
processStatusChanged, sigCHLD  :: Signal
realTimeAlarm, sigALRM         :: Signal
segmentationViolation, sigSEGV :: Signal
softwareStop, sigSTOP          :: Signal
softwareTermination, sigTERM   :: Signal
userDefinedSignal1, sigUSR1    :: Signal
userDefinedSignal2, sigUSR2    :: Signal

signalProcess :: Signal -> ProcessID -> IO ()
```

`signalProcess int pid` calls `kill` to signal process `pid` with interrupt signal `int`.

```
raiseSignal :: Signal -> IO ()
```

`raiseSignal int` calls `kill` to signal the current process with interrupt signal `int`.

```
signalProcessGroup :: Signal -> ProcessGroupID -> IO ()
```

`signalProcessGroup int pgid` calls `kill` to signal all processes in group `pgid` with interrupt signal `int`.

```
setStoppedChildFlag :: Bool -> IO Bool
```

`setStoppedChildFlag bool` sets a flag which controls whether or not the `NOCLDSTOP` option will be used the next time a signal handler is installed for `SIGCHLD`. If `bool` is `True` (the default), `NOCLDSTOP` will not be used; otherwise it will be. The operation never fails.

```
queryStoppedChildFlag :: IO Bool
```

`queryStoppedChildFlag` queries the flag which controls whether or not the `NOCLDSTOP` option will be used the next time a signal handler is installed for `SIGCHLD`. If `NOCLDSTOP` will be used, it returns `False`; otherwise (the default) it returns `True`. The operation never fails.

```
emptySignalSet :: SignalSet
fullSignalSet  :: SignalSet
addSignal      :: Signal -> SignalSet -> SignalSet
deleteSignal   :: Signal -> SignalSet -> SignalSet
inSignalSet    :: Signal -> SignalSet -> Bool

installHandler :: Signal
               -> Handler
               -> Maybe SignalSet       - other signals to block
               -> IO Handler            - old handler
```

`installHandler int handler iset` calls `sigaction` to install an interrupt handler for signal `int`. If `handler` is `Default`, `SIG_DFL` is installed; if `handler` is `Ignore`, `SIG_IGN` is installed; if `handler` is `Catch action`, a handler is installed which will invoke `action` in a new thread when (or shortly after) the signal is received. See Section 3.1 for details on how to communicate between threads.

If `iset` is `Just s`, then the `sa_mask` of the `sigaction` structure is set to `s`; otherwise it is cleared. The previously installed signal handler for `int` is returned.

```
getSignalMask :: IO SignalSet
```

`getSignalMask` calls `sigprocmask` to determine the set of interrupts which are currently being blocked.

```
setSignalMask :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_SETMASK` to block all interrupts in `mask`. The previous set of blocked interrupts is returned.

```
blockSignals :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_BLOCK` to add all interrupts in `mask` to the set of blocked interrupts. The previous set of blocked interrupts is returned.

```
unBlockSignals :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_UNBLOCK` to remove all interrupts in `mask` from the set of blocked interrupts. The previous set of blocked interrupts is returned.

```
getPendingSignals :: IO SignalSet
```

`getPendingSignals` calls `sigpending` to obtain the set of interrupts which have been received but are currently blocked.

```
awaitSignal :: Maybe SignalSet -> IO ()
```

`awaitSignal iset` suspends execution until an interrupt is received. If `iset` is `Just s`, `awaitSignal` calls `sigsuspend`, installing `s` as the new signal mask before suspending execution; otherwise, it calls `pause`. `awaitSignal` returns on receipt of a signal. If you have installed any signal handlers with `installHandler`, it may be wise to call `yield` directly after `awaitSignal` to ensure that the signal handler runs as promptly.

```
scheduleAlarm :: Int -> IO Int
```

`scheduleAlarm i` calls `alarm` to schedule a real time alarm at least `i` seconds in the future.

```
sleep :: Int -> IO ()
```

`sleep i` calls `sleep` to suspend execution of the program until at least `i` seconds have elapsed or a signal is received.

8.3. Posix Process Environment

```
getProcessID :: IO ProcessID
```

`getProcessID` calls `getpid` to obtain the `ProcessID` for the current process.

```
getParentProcessID :: IO ProcessID
```

`getProcessID` calls `getppid` to obtain the `ProcessID` for the parent of the current process.

```
getRealUserID :: IO UserID
```

`getRealUserID` calls `getuid` to obtain the real `UserID` associated with the current process.

```
getEffectiveUserID :: IO UserID
```

`getEffectiveUserID` calls `geteuid` to obtain the effective `UserID` associated with the current process.

```
setUserID :: UserID -> IO ()
```

`setUserID uid` calls `setuid` to set the real, effective, and saved set-user-id associated with the current process to `uid`.

```
getLoginName :: IO String
```

`getLoginName` calls `getlogin` to obtain the login name associated with the current process.

```
getRealGroupID :: IO GroupID
```

`getRealGroupID` calls `getgid` to obtain the real `GroupID` associated with the current process.

```
getEffectiveGroupID :: IO GroupID
```

`getEffectiveGroupID` calls `getegid` to obtain the effective `GroupID` associated with the current process.

```
setGroupID :: GroupID -> IO ()
```

`setGroupID gid` calls `setgid` to set the real, effective, and saved set-group-id associated with the current process to `gid`.

```
getGroups :: IO [GroupID]
```

`getGroups` calls `getgroups` to obtain the list of supplementary `GroupID`s associated with the current process.

```
getEffectiveUserName :: IO String
```

`getEffectiveUserName` calls `cuserid` to obtain a name associated with the effective `UserID` of the process.

```
getProcessGroupID :: IO ProcessGroupID
```

`getProcessGroupID` calls `getpgrp` to obtain the `ProcessGroupID` for the current process.

```
createProcessGroup :: ProcessID -> IO ProcessGroupID
```

`createProcessGroup pid` calls `setpgid` to make process `pid` a new process group leader.

```
joinProcessGroup :: ProcessGroupID -> IO ProcessGroupID
```

`joinProcessGroup pgid` calls `setpgid` to set the `ProcessGroupID` of the current process to `pgid`.

```
setProcessGroupID :: ProcessID -> ProcessGroupID -> IO ()
```

`setProcessGroupID pid pgid` calls `setpgid` to set the `ProcessGroupID` for process `pid` to `pgid`.

```
createSession :: IO ProcessGroupID
```

`createSession` calls `setsid` to create a new session with the current process as session leader.

```
systemName :: SystemID -> String
nodeName :: SystemID -> String
release :: SystemID -> String
version :: SystemID -> String
machine :: SystemID -> String

getSystemID :: IO SystemID
```

`getSystemID` calls `uname` to obtain information about the current operating system.

```
> epochTime :: IO EpochTime
```

`epochTime` calls `time` to obtain the number of seconds that have elapsed since the epoch (Jan 01 00:00:00 GMT 1970).

```
elapsedTime     :: ProcessTimes -> ClockTick
userTime        :: ProcessTimes -> ClockTick
systemTime      :: ProcessTimes -> ClockTick
childUserTime   :: ProcessTimes -> ClockTick
childSystemTime :: ProcessTimes -> ClockTick

getProcessTimes :: IO ProcessTimes
```

`getProcessTimes` calls `times` to obtain time-accounting information for the current process and its children.

```
getControllingTerminalName :: IO FilePath
```

`getControllingTerminalName` calls `ctermid` to obtain a name associated with the controlling terminal for the process. If a controlling terminal exists, `getControllingTerminalName` returns the name of the controlling terminal.

The operation may fail with:

`NoSuchThing`

  There is no controlling terminal, or its name cannot be determined.

`SystemError`

  Various other causes.

```
getTerminalName :: Fd -> IO FilePath
```

`getTerminalName fd` calls `ttyname` to obtain a name associated with the terminal for `Fd fd`. If `fd` is associated with a terminal, `getTerminalName` returns the name of the terminal.

The operation may fail with:

`InappropriateType`

  The channel is not associated with a terminal.

`NoSuchThing`

  The channel is associated with a terminal, but it has no name.

`SystemError`

  Various other causes.

```
queryTerminal :: Fd -> IO Bool
```

`queryTerminal fd` calls `isatty` to determine whether or not `Fd fd` is associated with a terminal.

```
getSysVar :: SysVar -> IO Limit
```

`getSysVar var` calls `sysconf` to obtain the dynamic value of the requested configurable system limit or option. For defined system limits, `getSysVar` returns the associated value. For defined system options, the result of `getSysVar` is undefined, but not failure.

The operation may fail with:

`NoSuchThing`

The requested system limit or option is undefined.

8.4. Posix operations on files and directories

```
openDirStream :: FilePath -> IO DirStream
```

`openDirStream dir` calls `opendir` to obtain a directory stream for `dir`.

```
readDirStream :: DirStream -> IO String
```

`readDirStream dp` calls `readdir` to obtain the next directory entry (`struct dirent`) for the open directory stream `dp`, and returns the `d_name` member of that structure.

The operation may fail with:

`EOF`

End of file has been reached.

`SystemError`

Various other causes.

```
rewindDirStream :: DirStream -> IO ()
```

`rewindDirStream dp` calls `rewinddir` to reposition the directory stream `dp` at the beginning of the directory.

```
closeDirStream :: DirStream -> IO ()
```

`closeDirStream dp` calls `closedir` to close the directory stream `dp`.

```
getWorkingDirectory :: IO FilePath
```

`getWorkingDirectory` calls `getcwd` to obtain the name of the current working directory.

```
changeWorkingDirectory :: FilePath -> IO ()
```

`changeWorkingDirectory dir` calls `chdir` to change the current working directory to `dir`.

```
nullFileMode       :: FileMode       - ------
ownerReadMode      :: FileMode       - r-----
ownerWriteMode     :: FileMode       - -w-----
ownerExecuteMode   :: FileMode       - -x----
groupReadMode      :: FileMode       - --r---
groupWriteMode     :: FileMode       - ---w---
groupExecuteMode   :: FileMode       - ---x--
otherReadMode      :: FileMode       - ----r-
otherWriteMode     :: FileMode       - -----w-
otherExecuteMode   :: FileMode       - -----x
setUserIDMode      :: FileMode       - -S----
setGroupIDMode     :: FileMode       - ---S--

stdFileMode        :: FileMode       - rw-rw-rw-

ownerModes         :: FileMode       - rwx----
groupModes         :: FileMode       - --rwx--
otherModes         :: FileMode       - ----rwx
accessModes        :: FileMode       - rwxrwxrwx

unionFileModes     :: FileMode -> FileMode -> FileMode
intersectFileModes :: FileMode -> FileMode -> FileMode

stdInput  :: Fd
stdInput  = intToFd 0

stdOutput :: Fd
stdOutput = intToFd 1

stdError  :: Fd
stdError  = intToFd 2

data OpenFileFlags =
 OpenFileFlags {
    append    :: Bool,
    exclusive :: Bool,
    noctty    :: Bool,
    nonBlock  :: Bool,
    trunc     :: Bool
 }

openFd :: FilePath
       -> OpenMode
       -> Maybe FileMode  - Just x => O_CREAT, Nothing => must exist
       -> OpenFileFlags
       -> IO Fd
```

`openFd path acc mode (OpenFileFlags app excl noctty nonblock trunc)` calls `open` to obtain a `Fd` for the file `path` with access mode `acc`. If `mode` is `Just m`, the `O_CREAT` flag is set and the file's permissions will be based on `m` if it does not already exist; otherwise, the `O_CREAT` flag is not set. The arguments `app`, `excl`, `noctty`, `nonblock`, and `trunc` control whether or not the flags `O_APPEND`, `O_EXCL`, `O_NOCTTY`, `O_NONBLOCK`, and `O_TRUNC` are set, respectively.

```
createFile :: FilePath -> FileMode -> IO Fd
```

`createFile path mode` calls `creat` to obtain a `Fd` for file `path`, which will be created with permissions based on `mode` if it does not already exist.

```
setFileCreationMask :: FileMode -> IO FileMode
```

`setFileCreationMask mode` calls `umask` to set the process's file creation mask to `mode`. The previous file creation mask is returned.

```
createLink :: FilePath -> FilePath -> IO ()
```

`createLink old new` calls `link` to create a new path, `new`, linked to an existing file, `old`.

```
createDirectory :: FilePath -> FileMode -> IO ()
```

`createDirectory dir mode` calls `mkdir` to create a new directory, `dir`, with permissions based on `mode`.

```
createNamedPipe :: FilePath -> FileMode -> IO ()
```

`createNamedPipe fifo mode` calls `mkfifo` to create a new named pipe, `fifo`, with permissions based on `mode`.

```
removeLink :: FilePath -> IO ()
```

`removeLink path` calls `unlink` to remove the link named `path`.

```
removeDirectory :: FilePath -> IO ()
```

`removeDirectory dir` calls `rmdir` to remove the directory named `dir`.

```
rename :: FilePath -> FilePath -> IO ()
```

`rename old new` calls `rename` to rename a file or directory from `old` to `new`.

```
fileMode          :: FileStatus -> FileMode

fileID            :: FileStatus -> FileID
```

```
deviceID          :: FileStatus -> DeviceID

linkCount         :: FileStatus -> LinkCount

fileOwner         :: FileStatus -> UserID
fileGroup         :: FileStatus -> GroupID
fileSize          :: FileStatus -> FileOffset

accessTime        :: FileStatus -> EpochTime
modificationTime  :: FileStatus -> EpochTime
statusChangeTime  :: FileStatus -> EpochTime

isDirectory       :: FileStatus -> Bool
isCharacterDevice :: FileStatus -> Bool
isBlockDevice     :: FileStatus -> Bool
isRegularFile     :: FileStatus -> Bool
isNamedPipe       :: FileStatus -> Bool

getFileStatus     :: FilePath -> IO FileStatus
```

`getFileStatus path` calls `stat` to get the `FileStatus` information for the file `path`.

```
getFdStatus :: Fd -> IO FileStatus
```

`getFdStatus fd` calls `fstat` to get the `FileStatus` information for the file associated with `Fd` `fd`.

```
queryAccess :: FilePath -> Bool -> Bool -> Bool -> IO Bool
```

`queryAccess path r w x` calls `access` to test the access permissions for file `path`. The three arguments, `r`, `w`, and `x` control whether or not `access` is called with `R_OK`, `W_OK`, and `X_OK` respectively.

```
queryFile :: FilePath -> IO Bool
```

`queryFile path` calls `access` with `F_OK` to test for the existence for file `path`.

```
setFileMode :: FilePath -> FileMode -> IO ()
```

`setFileMode path mode` calls `chmod` to set the permission bits associated with file `path` to `mode`.

```
setOwnerAndGroup :: FilePath -> UserID -> GroupID -> IO ()
```

`setOwnerAndGroup path uid gid` calls `chown` to set the `UserID` and `GroupID` associated with file `path` to `uid` and `gid`, respectively.

```
setFileTimes :: FilePath -> EpochTime -> EpochTime -> IO ()
```

`setFileTimes path atime mtime` calls `utime` to set the access and modification times associated with file `path` to `atime` and `mtime`, respectively.

```
touchFile :: FilePath -> IO ()
```

`touchFile path` calls `utime` to set the access and modification times associated with file `path` to the current time.

```
getPathVar :: PathVar -> FilePath -> IO Limit
```

`getPathVar var path` calls `pathconf` to obtain the dynamic value of the requested configurable file limit or option associated with file or directory `path`. For defined file limits, `getPathVar` returns the associated value. For defined file options, the result of `getPathVar` is undefined, but not failure. The operation may fail with:

`NoSuchThing`

The requested file limit or option is undefined.

`SystemError`

Various other causes.

```
getFdVar :: PathVar -> Fd -> IO Limit
```

`getFdVar var fd` calls `fpathconf` to obtain the dynamic value of the requested configurable file limit or option associated with the file or directory attached to the open channel `fd`. For defined file limits, `getFdVar` returns the associated value. For defined file options, the result of `getFdVar` is undefined, but not failure.

The operation may fail with:

`NoSuchThing`

The requested file limit or option is undefined.

`SystemError`

Various other causes.

8.5. Posix Input and Output Primitives

```
createPipe :: IO (Fd, Fd)
```

`createPipe` calls `pipe` to create a pipe and returns a pair of `Fd`s, the first for reading and the second for writing.

```
dup :: Fd -> IO Fd
```

`dup fd` calls `dup` to duplicate `Fd fd` to another `Fd`.

```
dupTo :: Fd -> Fd -> IO ()
```

`dupTo src dst` calls `dup2` to duplicate `Fd src` to `Fd dst`.

```
fdClose :: Fd -> IO ()
```

`fdClose fd` calls `close` to close `Fd fd`.

```
fdRead :: Fd -> ByteCount -> IO (String, ByteCount)
```

`fdRead fd nbytes` calls `read` to read at most `nbytes` bytes from `Fd fd`, and returns the result as a string paired with the number of bytes actually read.

The operation may fail with:

`EOF`

End of file has been reached.

`SystemError`

Various other causes.

```
fdWrite :: Fd -> String -> IO ByteCount
```

`fdWrite fd s` calls `write` to write the string `s` to `Fd fd` as a contiguous sequence of bytes. It returns the number of bytes successfully written.

```
queryFdOption :: FdOption -> Fd -> IO Bool
```

`getFdOption opt fd` calls `fcntl` to determine whether or not the flag associated with `FdOption opt` is set for `Fd fd`.

```
setFdOption :: Fd -> FdOption -> Bool -> IO ()
```

`setFdOption fd opt val` calls `fcntl` to set the flag associated with `FdOption` `opt` on `Fd` `fd` to `val`.

```
getLock :: Fd -> FileLock -> IO (Maybe (ProcessID, FileLock))
```

`getLock fd lock` calls `fcntl` to get the first `FileLock` for `Fd` `fd` which blocks the `FileLock` `lock`. If no such `FileLock` exists, `getLock` returns `Nothing`. Otherwise, it returns `Just (pid, block)`, where `block` is the blocking `FileLock` and `pid` is the `ProcessID` of the process holding the blocking `FileLock`.

```
setLock :: Fd -> FileLock -> IO ()
```

`setLock fd lock` calls `fcntl` with `F_SETLK` to set or clear a lock segment for `Fd` `fd` as indicated by the `FileLock` `lock`. `setLock` does not block, but fails with `SystemError` if the request cannot be satisfied immediately.

```
waitToSetLock :: Fd -> FileLock -> IO ()
```

`waitToSetLock fd lock` calls `fcntl` with `F_SETLKW` to set or clear a lock segment for `Fd` `fd` as indicated by the `FileLock` `lock`. If the request cannot be satisfied immediately, `waitToSetLock` blocks until the request can be satisfied.

```
fdSeek :: Fd -> SeekMode -> FileOffset -> IO FileOffset
```

`fdSeek fd whence offset` calls `lseek` to position the `Fd` `fd` at the given `offset` from the starting location indicated by `whence`. It returns the resulting offset from the start of the file in bytes.

8.6. Posix, Device- and Class-Specific Functions

```
terminalMode    :: TerminalMode -> TerminalAttributes -> Bool
withMode        :: TerminalAttributes -> TerminalMode -
> TerminalAttributes
withoutMode     :: TerminalAttributes -> TerminalMode -
> TerminalAttributes

bitsPerByte     :: TerminalAttributes -> Int
withBits        :: TerminalAttributes -> Int -> TerminalAttributes

controlChar     :: TerminalAttributes -> ControlCharacter -> Maybe Char
withCC          :: TerminalAttributes
                -> (ControlCharacter, Char)
                -> TerminalAttributes
```

```
withoutCC        :: TerminalAttributes
                 -> ControlCharacter
                 -> TerminalAttributes

inputTime        :: TerminalAttributes -> Int
withTime         :: TerminalAttributes -> Int -> TerminalAttributes

minInput         :: TerminalAttributes -> Int
withMinInput     :: TerminalAttributes -> Int -> TerminalAttributes

inputSpeed       :: TerminalAttributes -> BaudRate
withInputSpeed   :: TerminalAttributes -> BaudRate -> TerminalAttributes

outputSpeed      :: TerminalAttributes -> BaudRate
withOutputSpeed  :: TerminalAttributes -> BaudRate -> TerminalAttributes

getTerminalAttributes :: Fd -> IO TerminalAttributes
```

`getTerminalAttributes fd` calls `tcgetattr` to obtain the `TerminalAttributes` associated with `Fd` `fd`.

```
setTerminalAttributes :: Fd
                      -> TerminalAttributes
                      -> TerminalState
                      -> IO ()
```

`setTerminalAttributes fd attr ts` calls `tcsetattr` to change the `TerminalAttributes` associated with `Fd` `fd` to `attr`, when the terminal is in the state indicated by `ts`.

```
sendBreak :: Fd -> Int -> IO ()
```

`sendBreak fd duration` calls `tcsendbreak` to transmit a continuous stream of zero-valued bits on `Fd` `fd` for the specified implementation-dependent `duration`.

```
drainOutput :: Fd -> IO ()
```

`drainOutput fd` calls `tcdrain` to block until all output written to `Fd` `fd` has been transmitted.

```
discardData :: Fd -> QueueSelector -> IO ()
```

`discardData fd queues` calls `tcflush` to discard pending input and/or output for `Fd` `fd`, as indicated by the `QueueSelector` `queues`.

```
controlFlow :: Fd -> FlowAction -> IO ()
```

`controlFlow fd action` calls `tcflow` to control the flow of data on `Fd fd`, as indicated by `action`.

```
getTerminalProcessGroupID :: Fd -> IO ProcessGroupID
```

`getTerminalProcessGroupID fd` calls `tcgetpgrp` to obtain the `ProcessGroupID` of the foreground process group associated with the terminal attached to `Fd fd`.

```
setTerminalProcessGroupID :: Fd -> ProcessGroupID -> IO ()
```

`setTerminalProcessGroupID fd pgid` calls `tcsetpgrp` to set the `ProcessGroupID` of the foreground process group associated with the terminal attached to `Fd fd` to `pgid`.

8.7. Posix System Databases

```
groupName    :: GroupEntry -> String
groupID      :: GroupEntry -> GroupID
groupMembers :: GroupEntry -> [String]
```

```
getGroupEntryForID :: GroupID -> IO GroupEntry
```

`getGroupEntryForID gid` calls `getgrgid` to obtain the `GroupEntry` information associated with `GroupID gid`.

The operation may fail with:

`NoSuchThing`

There is no group entry for the GroupID.

```
getGroupEntryForName :: String -> IO GroupEntry
```

`getGroupEntryForName name` calls `getgrnam` to obtain the `GroupEntry` information associated with the group called `name`.

The operation may fail with:

`NoSuchThing`

There is no group entry for the name.

```
userName      :: UserEntry -> String
```

```
userID        :: UserEntry -> UserID
userGroupID   :: UserEntry -> GroupID
homeDirectory :: UserEntry -> String
userShell     :: UserEntry -> String

getUserEntryForID :: UserID -> IO UserEntry
```

`getUserEntryForID gid` calls `getpwuid` to obtain the `UserEntry` information associated with `UserID uid`. The operation may fail with:

`NoSuchThing`

There is no user entry for the UserID.

```
getUserEntryForName :: String -> IO UserEntry
```

`getUserEntryForName name` calls `getpwnam` to obtain the `UserEntry` information associated with the user login `name`.

The operation may fail with:

`NoSuchThing`

There is no user entry for the name.

8.8. POSIX Errors

```
getErrorCode :: IO ErrorCode
```

`getErrorCode` returns the current value of the external variable `errno`. It never fails.

```
setErrorCode :: ErrorCode -> IO ()
```

`setErrorCode err` sets the external variable `errno` to `err`. It never fails.

```
noError :: ErrorCode
noError = 0

argumentListTooLong, e2BIG            :: ErrorCode
badFd, eBADF                          :: ErrorCode
brokenPipe, ePIPE                     :: ErrorCode
directoryNotEmpty, eNOTEMPTY          :: ErrorCode
```

```
execFormatError, eNOEXEC                  :: ErrorCode
fileAlreadyExists, eEXIST                 :: ErrorCode
fileTooLarge, eFBIG                       :: ErrorCode
filenameTooLong, eNAMETOOLONG             :: ErrorCode
improperLink, eXDEV                       :: ErrorCode
inappropriateIOControlOperation, eNOTTY :: ErrorCode
inputOutputError, eIO                     :: ErrorCode
interruptedOperation, eINTR               :: ErrorCode
invalidArgument, eINVAL                   :: ErrorCode
invalidSeek, eSPIPE                       :: ErrorCode
isADirectory, eISDIR                      :: ErrorCode
noChildProcess, eCHILD                    :: ErrorCode
noLocksAvailable, eNOLCK                  :: ErrorCode
noSpaceLeftOnDevice, eNOSPC               :: ErrorCode
noSuchOperationOnDevice, eNODEV           :: ErrorCode
noSuchDeviceOrAddress, eNXIO              :: ErrorCode
noSuchFileOrDirectory, eNOENT             :: ErrorCode
noSuchProcess, eSRCH                      :: ErrorCode
notADirectory, eNOTDIR                    :: ErrorCode
notEnoughMemory, eNOMEM                   :: ErrorCode
operationNotImplemented, eNOSYS           :: ErrorCode
operationNotPermitted, ePERM              :: ErrorCode
permissionDenied, eACCES                  :: ErrorCode
readOnlyFileSystem, eROFS                 :: ErrorCode
resourceBusy, eBUSY                       :: ErrorCode
resourceDeadlockAvoided, eDEADLK          :: ErrorCode
resourceTemporarilyUnavailable, eAGAIN   :: ErrorCode
tooManyLinks, eMLINK                      :: ErrorCode
tooManyOpenFiles, eMFILE                  :: ErrorCode
tooManyOpenFilesInSystem, eNFILE          :: ErrorCode
```

Chapter 9. The `text` package: text manipulation

9.1. `HaXml`: Handling XML data

HaXml is a library for converting Haskell data structures into XML and vice versa. The documentation hasn't been incorporated into this book as yet, but for now it can be found at The HaXml page (http://www.cs.york.ac.uk/fp/HaXml/).

9.2. `MatchPS`: The Perl-like matching interface

(Sigbjørn Finne supplied the regular-expressions interface.)

The `MatchPS` module provides Perl-like "higher-level" facilities to operate on `PackedStrings` (Section 5.28). The regular expressions in question are in Perl syntax. The "flags" on various functions can include: `i` for case-insensitive, `s` for single-line mode, and `g` for global. (It's probably worth your time to peruse the source code. . .)

```
matchPS :: PackedString    - regexp
        -> PackedString    - string to match
        -> [Char]          - flags
        -> Maybe REmatch   - info about what matched and where

searchPS :: PackedString    - regexp
         -> PackedString    - string to match
         -> [Char]          - flags
         -> Maybe REmatch

- Perl-like match-and-substitute:
substPS :: PackedString     - regexp
        -> PackedString     - replacement
        -> [Char]           - flags
        -> PackedString     - string
        -> PackedString

- same as substPS, but no prefix and suffix:
replacePS :: PackedString  - regexp
          -> PackedString  - replacement
          -> [Char]        - flags
          -> PackedString  - string
          -> PackedString
```

```
match2PS :: PackedString   - regexp
         -> PackedString   - string1 to match
         -> PackedString   - string2 to match
         -> [Char]         - flags
         -> Maybe REmatch

search2PS :: PackedString  - regexp
          -> PackedString  - string to match
          -> PackedString  - string to match
          -> [Char]        - flags
          -> Maybe REmatch

- functions to pull the matched pieces out of an REmatch:

getMatchesNo    :: REmatch -> Int
getMatchedGroup :: REmatch -> Int -> PackedString -> PackedString
getWholeMatch   :: REmatch -> PackedString -> PackedString
getLastMatch    :: REmatch -> PackedString -> PackedString
getAfterMatch   :: REmatch -> PackedString -> PackedString

- (reverse) brute-force string matching;
- Perl equivalent is index/rindex:
findPS, rfindPS :: PackedString -> PackedString -> Maybe Int

- Equivalent to Perl "chop" (off the last character, if any):
chopPS :: PackedString -> PackedString

- matchPrefixPS: tries to match as much as possible of strA starting
- from the beginning of strB (handy when matching fancy literals in
- parsers):
matchPrefixPS :: PackedString -> PackedString -> Int
```

9.3. `Parsec`: Parsing combinators

Parsec is a parsing combinator library. The documentation hasn't been incorporated into this book as yet, but for now it can be found at The Parsec page (http://www.cs.uu.nl/~daan/parsec.html).

9.4. `Pretty`: Pretty printing combimators

This library contains Simon Peyton Jones' implementation of John Hughes's pretty printer combinators.

```
infixl 6 <>, <+>
```

```
infixl 5 $$, $+$
data Doc   - the document datatype, abstract, instance of Show

- primitive documents
empty              :: Doc
semi, comma, colon :: Doc
space, equals      :: Doc
lparen, rparen     :: Doc
lbrack, rbrack     :: Doc
lbrace, rbrace     :: Doc

- converting values into documents
text     :: String   -> Doc
ptext    :: String   -> Doc
char     :: Char     -> Doc
int      :: Int      -> Doc
integer  :: Integer  -> Doc
float    :: Float    -> Doc
double   :: Double   -> Doc
rational :: Rational -> Doc

- wrapping documents into delimiters
parens, brackets, braces :: Doc -> Doc
quotes, doubleQuotes     :: Doc -> Doc

- combining documents
(<>)      :: Doc -> Doc -> Doc  - Beside
hcat      :: [Doc] -> Doc       - List version of <>
(<+>)     :: Doc -> Doc -> Doc  - Beside, separated by space
hsep      :: [Doc] -> Doc       - List version of <+>
($$)      :: Doc -> Doc -> Doc  - Above; "dovetails" if no overlap
vcat      :: [Doc] -> Doc       - List version of $$
($+$)     :: Doc -> Doc -> Doc  - Above; never overlaps
cat       :: [Doc] -> Doc       - Either hcat or vcat
sep       :: [Doc] -> Doc       - Either hsep or vcat
fcat      :: [Doc] -> Doc       - "Paragraph fill" version of cat
fsep      :: [Doc] -> Doc       - "Paragraph fill" version of sep
nest      :: Int -> Doc -> Doc  - Nested
hang      :: Doc -> Int -> Doc -> Doc
punctuate :: Doc -> [Doc] -> [Doc]
- punctuate p [d1, ... dn] = [d1 <> p, d2 <> p, ... dn-1 <> p, dn]

- default rendering (normal mode, line length 100, 1.5 ribbons per line)
render :: Doc -> String

- rendering style:
data Style
 = Style { mode           - rendering mode
         , lineLength     - in chars
```

```
 , ribbonsPerLine - Ratio of ribbon to line length.
         }

- default rendering style
style :: Style
style = Style { mode            = PageMode
              , lineLength      = 100
      , ribbonsPerLine  = 1.5
      }

renderStyle :: Style -> Doc -> String

- general rendering of documents
fullRender ::
      Mode
   -> Int                        - Line length
   -> Float                      - Ribbons per line
   -> (TextDetails -> a -> a)    - What to do with text
   -> a                          - What to do at the end
   -> Doc                        - The document to render
   -> a                          - Result

data Mode =
     PageMode                    - Normal
   | ZigZagMode                  - With zig-zag cuts
   | LeftMode                    - No indentation, infinitely long lines
   | OneLineMode                 - All on one line

data TextDetails =
     Chr  Char
   | Str  String
   | PStr String

- predicate on documents
isEmpty :: Doc -> Bool
```

9.5. `Regex`: The low-level regex matching interface

(Sigbjørn Finne supplied the regular-expressions interface.)

The `Regex` library provides quite direct interface to the GNU regular-expression library, for doing manipulation on `PackedStrings` (Section 5.28). You probably need to see the GNU documentation if you are operating at this level. Alternatively, you can use the simpler and higher-level `RegexString` (Section 9.6) interface.

The datatypes and functions that `Regex` provides are:

```
data PatBuffer  # just a bunch of bytes (mutable)

data REmatch
 = REmatch (Array Int GroupBounds)  - for $1, ... $n
           GroupBounds              - for $` (everything before match)
           GroupBounds              - for $& (entire matched string)
           GroupBounds              - for $' (everything after)
           GroupBounds              - for $+ (matched by last bracket)

- GroupBounds hold the interval where a group
- matched inside a string, e.g.
-
- matching "reg(exp)" "a regexp" returns the pair (5,7) for the
- (exp) group. (PackedString indices start from 0)

type GroupBounds = (Int, Int)

re_compile_pattern
        :: PackedString         - pattern to compile
        -> Bool                 - True <=> assume single-line mode
        -> Bool                 - True <=> case-insensitive
        -> IO PatBuffer

re_match :: PatBuffer           - compiled regexp
         -> PackedString        - string to match
         -> Int                 - start position
         -> Bool                - True <=> record results in registers
         -> IO (Maybe REmatch)

- Matching on 2 strings is useful when you're dealing with multiple
- buffers, which is something that could prove useful for
- PackedStrings, as we don't want to stuff the contents of a file
- into one massive heap chunk, but load (smaller chunks) on demand.

re_match2 :: PatBuffer          - 2-string version
          -> PackedString
          -> PackedString
          -> Int
          -> Int
          -> Bool
          -> IO (Maybe REmatch)

re_search :: PatBuffer          - compiled regexp
          -> PackedString       - string to search
          -> Int                - start index
          -> Int                - stop index
          -> Bool               - True <=> record results in registers
          -> IO (Maybe REmatch)
```

```
re_search2 :: PatBuffer         - Double buffer search
           -> PackedString
           -> PackedString
           -> Int               - start index
           -> Int               - range (?)
           -> Int               - stop index
           -> Bool              - True <=> results in registers
           -> IO (Maybe REmatch)
```

9.6. `RegexString`: Regex matching made simple

For simple regular expression operations, the `Regex` library is a little heavyweight. `RegexString` permits regex matching on ordinary Haskell `String`s.

The datatypes and functions that `RegexString` provides are:

```
data Regex - a compiled regular expression

mkRegex
        :: String       - regexp to compile
        -> Regex        - compiled regexp

mkRegexWithOpts
        :: String       - regexp to compile
        -> Bool         - single line mode?
        -> Bool         - case insensitive?
        -> Regex

matchRegex
        :: Regex        - compiled regexp
        -> String       - string to match
        -> Maybe [String] - text of $1, $2, ... (if matched)

matchRegexAll
        :: Regex
        -> String
        -> Maybe ( String,  - before
                   String,  - entire
                   String,  - after
                   String,  - last bracket
                   [String] - $1..
                 )
```

`mkRegex` makes a regular expression with the default options (multi-line, case-sensitive), whereas `mkRegexWithOpts` takes the values for these options as parameteres. The syntax of regular expressions is otherwise that of Perl.

`matchRegex`performs a simple match given a pattern and a string, returning either `Nothing` if the match failed, or `Just [s1...sn]` if the match succeeded, where `s1...sn` are the bracketed components of the pattern in left-to-right order. Note that there may be more components in the returned list than were in the pattern, and that any component which either matched the empty string or wasn't matched at all (because it was part of an optional part of the pattern), will be empty.

`matchRegexAll` returns more information about the match, including the string before the matched portion and the string after the matched portion.

Chapter 10. The `util` package: miscellaneous utilities

10.1. `GetOpt`: Command line parsing

The `GetOpt` library contains Sven Panne's Haskell implementation of `getopt`, providing features nigh-on identical to GNU `getopt`:

```
module GetOpt where

- representing a single option:
data OptDescr a
 = Option [Char]         -    list of short option characters
          [String]       -    list of long option strings (without "-")
          (ArgDescr a)   -    argument descriptor
          String         -    explanation of option for user

- argument option:
data ArgDescr a
   = NoArg                  a         -    no argument expected
   | ReqArg (String       -> a) String -    option requires argument
   | OptArg (Maybe String -> a) String -    optional argument

usageInfo :: String          - header
          -> [OptDescr a]    - options recognised
          -> String          - nicely formatted decription of options

getOpt :: ArgOrder a    - non-option handling
       -> [OptDescr a]  - options recognised
       -> [String]      - the command-line
       -> ( [a]         - options
          , [String]    - non-options
  ,[String]     - error messages
  )

data ArgOrder a
  = RequireOrder
  | Permute
  | ReturnInOrder (String -> a)
```

- The command-line options recognised is described by a list of `OptDescr` values. The `OptDescr` describes the long and short strings that recognise the option, together with a help string and info on whether the option takes extra arguments, if any.

- From a list of option values, `usageInfo` returns a nicely formatted string that enumerates the different options supported together with a short message about what
- To decode a command-line with respect to a list of options, `getOpt` is used. It processes the command-line, and returns the list of values that matched (and those that didn't). The first argument to `getOpt` controls whether the user is to give the options in any old order or not.

To hopefully illuminate the role of the different `GetOpt` data structures, here's the command-line options for a (very simple) compiler:

```
module Opts where

import GetOpt
import Maybe ( fromMaybe )

data Flag
 = Verbose  | Version
 | Input String | Output String | LibDir String
   deriving Show

options :: [OptDescr Flag]
options =
 [ Option ['v']     ["verbose"] (NoArg Verbose)       "chatty out-
put on stderr"
 , Option ['V','?'] ["version"] (NoArg Version)       "show ver-
sion number"
 , Option ['o']     ["output"]  (OptArg outp "FILE")  "output FILE"
 , Option ['c']     []          (OptArg inp  "FILE")  "input FILE"
 , Option ['L']     ["libdir"]  (ReqArg LibDir "DIR") "library directory"
 ]

inp,outp :: Maybe String -> Flag
outp = Output . fromMaybe "stdout"
inp  = Input  . fromMaybe "stdout"

compilerOpts :: [String] -> IO ([Flag], [String])
compilerOpts argv =
   case (getOpt Permute options argv) of
      (o,n,[]  ) -> return (o,n)
      (_,_,errs) -> failIO (concat errs ++ usageInfo header options)
  where header = "Usage: ic [OPTION...] files..."
```

10.2. Memo: Fast memo functions

The `Memo` library provides fast polymorphic memo functions using hash tables. The interface is:

```
memo :: (a -> b) -> a -> b
```

So, for example, `memo f` is a version of `f` that caches the results of previous calls.

The searching is very fast, being based on pointer equality. One consequence of this is that the caching will only be effective if *exactly the same argument is passed again to the memoised function*. This means not just a copy of a previous argument, but the same instance. It's not useful to memoise integer functions using this interface, because integers are generally copied a lot and two instances of '27' are unlikely to refer to the same object.

This memoisation library works well when the keys are large (or even infinite).

The memo table implementation uses weak pointers and stable names (see the GHC/Hugs library document) to avoid space leaks and allow hashing for arbitrary Haskell objects. NOTE: while individual memo table entries will be garbage collected if the associated key becomes garbage, the memo table itself will not be collected if the function becomes garbage. We plan to fix this in a future version.

There's another version of `memo` if you want to explicitly give a size for the hash table (the default size is 1001 buckets):

```
memo_sized :: Int -> (a -> b) -> a -> b
```

10.3. `QuickCheck`

To do.

10.4. `Readline`: Command line editing

(Darren Moffat supplied the initial version of the `Readline` module.)

The `Readline` module is a straightforward interface to the GNU Readline library. As such, you will need to look at the GNU documentation (and have a `libreadline.a` file around somewhere...)

The main function you'll use is:

```
readline :: String{-the prompt-} -> IO (Maybe String)
```

If you want to mess around with Full Readline G(l)ory, we also provide:

```
type KeyCode = Char

type CallbackFunction =
    (Int ->     - Numeric Argument
```

```
     KeyCode -> - KeyCode of pressed Key
     IO Int)    - What's this?

initialize      :: IO ()
addHistory      :: String -> IO ()
bindKey         :: KeyCode -> CallbackFunction -> IO ()
addDefun        :: String  -> CallbackFunction -> Maybe KeyCode -> IO ()

getReadlineName :: IO String
setReadlineName :: String -> IO ()
getLineBuffer   :: IO String
setLineBuffer   :: String -> IO ()
getPoint        :: IO Int
setPoint        :: Int -> IO ()
getEnd          :: IO Int
setEnd          :: Int -> IO ()
getMark         :: IO Int
setMark         :: Int -> IO ()
setDone         :: Bool -> IO ()
setPendingInput :: KeyCode -> IO ()
getPrompt       :: IO String
getTerminalName :: IO String

inStream        :: Handle
outStream       :: Handle
```

(All those names are just Haskellised versions of what you will see in the GNU readline documentation.)

10.5. `Select`: Synchronous I/O multiplexing

The `Select` interface provides a Haskell wrapper for the `select()` OS call supplied by many modern UNIX variants. `Select` exports the following:

```
type TimeOut = Maybe Int
  - Nothing => wait indefinitely.
  - Just x | x >= 0    => block waiting for 'x' micro seconds.
  -        | otherwise => block waiting for '-x' micro seconds.

hSelect :: [Handle]
        -> [Handle]
        -> [Handle]
        -> TimeOut
        -> IO SelectResult

type SelectResult
```

```
 = ( [Handle]  - input  handles ready
   , [Handle]  - output handles ready
   , [Handle]  - exc.   handles ready
   )
```

Here's an example of how it could be used:

```
module Main(main) where

import Select
import IO

main :: IO ()
main = do
  hSetBuffering stdin NoBuffering
  putStrLn "waiting for input to appear"
  hSelect [stdin] [] [] Nothing
  putStrLn "input ready, let's try reading"
  x <- getChar
  print x
```

where the call to `hSelect` makes the process go to sleep until there's input available on `stdin`.

10.5.1. Using `hSelect` with Concurrent Haskell

In brief: don't. For two reasons:

- `hSelect` will cause all your Haskell threads to block until the `hSelect` returns, much like any call to a foreign function.
- You don't need to. Concurrent Haskell will let you do I/O on multiple file handles concurrently by forking threads, and if you need to assign a timeout, then this can be done using a combination of `threadDelay` (see Section 3.3) and asynchronous exceptions (see Section 5.12.7).

Chapter 11. The Win32 package

The win32 package is a thin and incomplete veneer over the Win32 API. Look at the source code to see what is available; the usage should be obvious from the Microsoft's C API documentation.