Building the Glasgow Functional Programming Tools Suite

The GHC Team

This guide is intended for people who want to build or modify programs from the Glasgow `fptools` suite (as distinct from those who merely want to *run* them). Installation instructions are now provided in the user guide.

The bulk of this guide applies to building on Unix systems; see Section 9 for Windows notes.

1. Getting the Glasgow `fptools` suite

Building the Glasgow tools *can* be complicated, mostly because there are so many permutations of what/why/how, e.g., “Build Happy with HBC, everything else with GHC, leave out profiling, and test it all on the ‘real’ NoFib programs.” Yeeps!

Happily, such complications don’t apply to most people. A few common “strategies” serve most purposes. Pick one and proceed as suggested:

Binary distribution.

> If your only purpose is to install some of the `fptools` suite then the easiest thing to do is to get a binary distribution. In the binary distribution everything is pre-compiled for your particular machine architecture and operating system, so all you should have to do is install the binaries and libraries in suitable places. The user guide describes how to do this.
>
> A binary distribution may not work for you for two reasons. First, we may not have built the suite for the particular architecture/OS platform you want. That may be due to lack of time and energy (in which case you can get a source distribution and build from it; see below). Alternatively, it may be because we haven’t yet ported the suite to your architecture, in which case you are considerably worse off.
>
> The second reason a binary distribution may not be what you want is if you want to read or modify the souce code.

Source distribution.

> You have a supported platform, but (a) you like the warm fuzzy feeling of compiling things yourself; (b) you want to build something “extra”—e.g., a set of libraries with strictness-analysis turned off; or (c) you want to hack on GHC yourself.

A source distribution contains complete sources for one or more projects in the `fptools` suite. Not only that, but the more awkward machine-independent steps are done for you. For example, if you don't have **happy** you'll find it convenient that the source distribution contains the result of running **happy** on the parser specifications. If you don't want to alter the parser then this saves you having to find and install **happy**. You will still need a working version of GHC (preferably version 4.08+) on your machine in order to compile (most of) the sources, however.

Build GHC from intermediate C `.hc` files:

You need a working GHC to use a source distribution. What if you don't have a working GHC? Then you have no choice but to "bootstrap" up from the intermediate C (`.hc`) files that we provide. Building GHC on an unsupported platform falls into this category. Please see Section 7.

Once you have built GHC, you can build the other Glasgow tools with it.

In theory, you can (could?) build GHC with another Haskell compiler (e.g., HBC). We haven't tried to do this for ages and it almost certainly doesn't work any more (for tedious reasons).

The CVS repository.

We make releases infrequently. If you want more up-to-the minute (but less tested) source code then you need to get access to our CVS repository.

All the `fptools` source code is held in a CVS repository. CVS is a pretty good source-code control system, and best of all it works over the network.

The repository holds source code only. It holds no mechanically generated files at all. So if you check out a source tree from CVS you will need to install every utility so that you can build all the derived files from scratch.

More information about our CVS repository is available in the fptools CVS Cheat Sheet (http://www.haskell.org/ghc/cvs-cheat-sheet.html).

If you are going to do any building from sources (either from a source distribution or the CVS repository) then you need to read all of this manual in detail.

2. Things to check before you start typing

Here's a list of things to check before you get started.

1. Disk space needed: About 40MB (one tenth of one hamburger's worth) of disk space for the most basic binary distribution of GHC; more for some platforms, e.g., Alphas. An extra "bundle" (e.g., concurrent Haskell libraries) might take you to up to one fifth of a hamburger. You'll need over 100MB (say, one fifth a hamburger's worth) if you need to build the basic stuff

from scratch. All of the above are *estimates* of disk-space needs. (Note: our benchmark hamburger is a standard Double Whopper with Cheese, with an RRP of UKP2.99.)

2. Use an appropriate machine, compilers, and things. SPARC boxes, and PCs running Linux, BSD (any variant), or Solaris are all fully supported. Win32 and HP boxes are in pretty good shape. DEC Alphas running OSF/1, Linux or some BSD variant, MIPS and AIX boxes will need some minimal porting effort before they work (as of 4.06). Section 3 gives the full run-down on ports or lack thereof.
3. Be sure that the "pre-supposed" utilities are installed. Section 4 elaborates.
4. If you have any problem when building or installing the Glasgow tools, please check the "known pitfalls" (Section 8). Also check the FAQ for the version you're building, which should be available from the relevant download page on the GHC web site (http://www.haskell.org/ghc/). If you feel there is still some shortcoming in our procedure or instructions, please report it. For GHC, please see the bug-reporting section of the GHC Users' Guide (separate document), to maximise the usefulness of your report. If in doubt, please send a message to <`glasgow-haskell-bugs@haskell.org`>.

3. What machines the Glasgow tools run on

The main question is whether or not the Haskell compiler (GHC) runs on your platform.

A "platform" is a architecture/manufacturer/operating-system combination, such as `sparc-sun-solaris2`. Other common ones are `alpha-dec-osf2`, `hppa1.1-hp-hpux9`, `i386-unknown-linux`, `i386-unknown-solaris2`, `i386-unknown-freebsd`, `i386-unknown-cygwin32`, `m68k-sun-sunos4`, `mips-sgi-irix5`, `sparc-sun-sunos4`, `sparc-sun-solaris2`, `powerpc-ibm-aix`.

Bear in mind that certain "bundles", e.g. parallel Haskell, may not work on all machines for which basic Haskell compiling is supported.

Some libraries may only work on a limited number of platforms; for example, a sockets library is of no use unless the operating system supports the underlying BSDisms.

3.1. What platforms the Haskell compiler (GHC) runs on

The GHC hierarchy of Porting Goodness: (a) Best is a native-code generator; (b) next best is a "registerised" port; (c) the bare minimum is an "unregisterised" port. ("Unregisterised" is so terrible that we won't say more about it).

We use Sparcs running Solaris 2.7 and x86 boxes running FreeBSD and Linux, so those are the best supported platforms, unsurprisingly.

Here's everything that's known about GHC ports. We identify platforms by their "canonical" CPU/Manufacturer/OS triple.

alpha-dec-{osf,linux,freebsd,openbsd,netbsd}:

Currently non-working. The last working version (osf[1-3]) is GHC 3.02. A small amount of porting effort will be required to get Alpha support into GHC 4.xx, but we don't have easy access to machines right now, and there hasn't been a massive demand for support, so Alphas remain unsupported for the time being. Please get in touch if you either need Alpha support and/or can provide access to boxes.

sparc-sun-sunos4:

Probably works with minor tweaks, hasn't been tested for a while.

sparc-sun-solaris2:

Fully supported, including native-code generator.

hppa1.1-hp-hpux (HP-PA boxes running HPUX 9.x)

Works registerised. No native-code generator.

i386-unknown-linux (PCs running Linux—ELF binary format):

GHC works registerised, has a native code generator. You *must* have GCC 2.7.x or later. NOTE about `glibc` versions: GHC binaries built on a system running `glibc 2.0` won't work on a system running `glibc 2.1`, and vice versa. In general, don't expect compatibility between `glibc` versions, even if the shared library version hasn't changed.

i386-unknown-{freebsd,netbsd,openbsd) (PCs running FreeBSD 2.2 or higher, NetBSD, and possibly OpenBSD):

GHC works registerised. These systems provide ready-built packages of GHC, so if you just need binaries you're better off just installing the package.

i386-unknown-cygwin32:

Fully supported under Win9x/NT, including a native code generator. Requires the `cygwin32` compatibility library and a healthy collection of GNU tools (i.e., gcc, GNU ld, bash etc.).

mips-sgi-irix5:

Port currently doesn't work, needs some minimal porting effort. As usual, we don't have access to machines and there hasn't been an overwhelming demand for this port, but feel free to get in touch.

powerpc-ibm-aix:

Port currently doesn't work, needs some minimal porting effort. As usual, we don't have access to machines and there hasn't been an overwhelming demand for this port, but feel free to get in touch.

Various other systems have had GHC ported to them in the distant past, including various Motorola 68k boxes. The 68k support still remains, but porting to one of these systems will certainly be a non-trivial task.

3.2. What machines the other tools run on

Unless you hear otherwise, the other tools work if GHC works.

4. Installing pre-supposed utilities

Here are the gory details about some utility programs you may need; **perl**, **gcc** and **happy** are the only important ones. (PVM is important if you're going for Parallel Haskell.) The **configure** script will tell you if you are missing something.

Perl:

You have to have Perl to proceed! It is pretty easy to install.

Perl 5 is required. For Win32 platforms, you should use the binary supplied in the InstallShield (copy it to `/bin`). The Cygwin-supplied Perl seems not to work.

Perl should be put somewhere so that it can be invoked by the `#!` script-invoking mechanism. The full pathname may need to be less than 32 characters long on some systems.

GNU C (**gcc**):

We recommend using GCC version 2.95.2 on all platforms. Failing that, version 2.7.2 is stable on most platforms. Earlier versions of GCC can be assumed not to work, and versions in between 2.7.2 and 2.95.2 (including **egcs**) have varying degrees of stability depending on the platform.

If your GCC dies with "internal error" on some GHC source file, please let us know, so we can report it and get things improved. (Exception: on iX86 boxes—you may need to fiddle with GHC's `-monly-N-regs` option; see the User's Guide)

Happy:

Happy is a parser generator tool for Haskell, and is used to generate GHC's parsers. Happy is written in Haskell, and is a project in the CVS repository (`fptools/happy`). It can be built from source, but bear in mind that you'll need GHC installed in order to build it. To avoid the chicken/egg problem, install a binary distribtion of either Happy or GHC to get started. Happy distributions are available from Happy's Web Page (http://www.haskell.org/happy/).

Autoconf:

GNU Autoconf is needed if you intend to build from the CVS sources, it is *not* needed if you just intend to build a standard source distribution.

Autoconf builds the **configure** script from `configure.in` and `aclocal.m4`. If you modify either of these files, you'll need **autoconf** to rebuild `configure`.

sed

You need a working **sed** if you are going to build from sources. The build-configuration stuff needs it. GNU sed version 2.0.4 is no good! It has a bug in it that is tickled by the build-configuration. 2.0.5 is OK. Others are probably OK too (assuming we don't create too elaborate configure scripts.)

One `fptools` project is worth a quick note at this point, because it is useful for all the others: `glafp-utils` contains several utilities which aren't particularly Glasgow-ish, but Occasionally Indispensable. Like **lndir** for creating symbolic link trees.

4.1. Tools for building parallel GHC (GPH)

PVM version 3:

PVM is the Parallel Virtual Machine on which Parallel Haskell programs run. (You only need this if you plan to run Parallel Haskell. Concurent Haskell, which runs concurrent threads on a uniprocessor doesn't need it.) Underneath PVM, you can have (for example) a network of workstations (slow) or a multiprocessor box (faster).

The current version of PVM is 3.3.11; we use 3.3.7. It is readily available on the net; I think I got it from `research.att.com`, in `netlib`.

A PVM installation is slightly quirky, but easy to do. Just follow the `Readme` instructions.

bash:

Sadly, the **gr2ps** script, used to convert "parallelism profiles" to PostScript, is written in Bash (GNU's Bourne Again shell). This bug will be fixed (someday).

4.2. Tools for building the Documentation

The following additional tools are required if you want to format the documentation that comes with the `fptools` projects:

DocBook:

All our documentation is written in SGML, using the DocBook DTD. Instructions on installing and configuring the DocBook tools are in the installation guide (in the GHC user guide).

TeX:

A decent TeX distribution is required if you want to produce printable documentation. We recomment teTeX, which includes just about everything you need.

4.3. Other useful tools

Flex:

This is a quite-a-bit-better-than-Lex lexer. Used to build a couple of utilities in `glafp-utils`. Depending on your operating system, the supplied **lex** may or may not work; you should get the GNU version.

5. Building from source

You've been rash enough to want to build some of the Glasgow Functional Programming tools (GHC, Happy, nofib, etc.) from source. You've slurped the source, from the CVS repository or from a source distribution, and now you're sitting looking at a huge mound of bits, wondering what to do next.

Gingerly, you type **make**. Wrong already!

This rest of this guide is intended for duffers like me, who aren't really interested in Makefiles and systems configurations, but who need a mental model of the interlocking pieces so that they can make them work, extend them consistently when adding new software, and lay hands on them gently when they don't work.

5.1. Your source tree

The source code is held in your *source tree*. The root directory of your source tree *must* contain the following directories and files:

- `Makefile`: the root Makefile.
- `mk/`: the directory that contains the main Makefile code, shared by all the `fptools` software.
- `configure.in`, `config.sub`, `config.guess`: these files support the configuration process.
- `install-sh`.

All the other directories are individual *projects* of the `fptools` system—for example, the Glasgow Haskell Compiler (`ghc`), the Happy parser generator (`happy`), the `nofib` benchmark suite, and so on. You can have zero or more of these. Needless to say, some of them are needed to build others.

The important thing to remember is that even if you want only one project (`happy`, say), you must have a source tree whose root directory contains `Makefile`, `mk/`, `configure.in`, and the project(s) you want (`happy/` in this case). You cannot get by with just the `happy/` directory.

5.2. Build trees

While you can build a system in the source tree, we don't recommend it. We often want to build multiple versions of our software for different architectures, or with different options (e.g. profiling). It's very desirable to share a single copy of the source code among all these builds.

So for every source tree we have zero or more *build trees*. Each build tree is initially an exact copy of the source tree, except that each file is a symbolic link to the source file, rather than being a copy of the source file. There are "standard" Unix utilities that make such copies, so standard that they go by different names: **lndir**, **mkshadowdir** are two (If you don't have either, the source distribution includes sources for the X11 **lndir**—check out `fptools/glafp-utils/lndir`). See Section 5.4 for a typical invocation.

The build tree does not need to be anywhere near the source tree in the file system. Indeed, one advantage of separating the build tree from the source is that the build tree can be placed in a non-backed-up partition, saving your systems support people from backing up untold megabytes of easily-regenerated, and rapidly-changing, gubbins. The golden rule is that (with a single exception—Section 5.3) *absolutely everything in the build tree is either a symbolic link to the source tree, or else is mechanically generated*. It should be perfectly OK for your build tree to vanish overnight; an hour or two compiling and you're on the road again.

You need to be a bit careful, though, that any new files you create (if you do any development work) are in the source tree, not a build tree!

Remember, that the source files in the build tree are *symbolic links* to the files in the source tree. (The build tree soon accumulates lots of built files like `Foo.o`, as well.) You can *delete* a source file from the build tree without affecting the source tree (though it's an odd thing to do). On the other hand, if you *edit* a source file from the build tree, you'll edit the source-tree file directly. (You can set up Emacs so that if you edit a source file from the build tree, Emacs will silently create an edited copy of the source file in the build tree, leaving the source file unchanged; but the danger is that you think you've edited the source file whereas actually all you've done is edit the build-tree copy. More commonly you do want to edit the source file.)

Like the source tree, the top level of your build tree must be (a linked copy of) the root directory of the `fptools` suite. Inside Makefiles, the root of your build tree is called `$(FPTOOLS_TOP)`. In the rest of this document path names are relative to `$(FPTOOLS_TOP)` unless otherwise stated. For example, the file `ghc/mk/target.mk` is actually `$(FPTOOLS_TOP)/ghc/mk/target.mk`.

5.3. Getting the build you want

When you build `fptools` you will be compiling code on a particular *host platform*, to run on a particular *target platform* (usually the same as the host platform). The difficulty is that there are minor differences between different platforms; minor, but enough that the code needs to be a bit different for each. There are some big differences too: for a different architecture we need to build GHC with a different native-code generator.

There are also knobs you can turn to control how the `fptools` software is built. For example, you might want to build GHC optimised (so that it runs fast) or unoptimised (so that you can compile it fast after you've modified it. Or, you might want to compile it with debugging on (so that extra consistency-checking code gets included) or off. And so on.

All of this stuff is called the *configuration* of your build. You set the configuration using a three-step process.

Step 1: get ready for configuration.

Change directory to `$(FPTOOLS_TOP)` and issue the command **autoconf** (with no arguments). This GNU program converts `$(FPTOOLS_TOP)/configure.in` to a shell script called `$(FPTOOLS_TOP)/configure`.

Some projects, including GHC, have their own configure script. If there's an `$(FPTOOLS_TOP)/<project>/configure.in`, then you need to run **autoconf** in that directory too.

Both these steps are completely platform-independent; they just mean that the human-written file (`configure.in`) can be short, although the resulting shell script, **configure**, and `mk/config.h.in`, are long.

In case you don't have **autoconf** we distribute the results, **configure**, and `mk/config.h.in`, with the source distribution. They aren't kept in the repository, though.

Step 2: system configuration.

Runs the newly-created **configure** script, thus:

```
./configure [args]
```

configure's mission is to scurry round your computer working out what architecture it has, what operating system, whether it has the `vfork` system call, where **yacc** is kept, whether **gcc** is available, where various obscure `#include` files are, whether it's a leap year, and what the systems manager had for lunch. It communicates these snippets of information in two ways:

- It translates `mk/config.mk.in` to `mk/config.mk`, substituting for things between "`@`" brackets. So, "`@HaveGcc@`" will be replaced by "`YES`" or "`NO`" depending on what **configure** finds. `mk/config.mk` is included by every Makefile (directly or indirectly), so the configuration information is thereby communicated to all Makefiles.

- It translates `mk/config.h.in` to `mk/config.h`. The latter is `#included` by various C programs, which can thereby make use of configuration information.

configure takes some optional arguments. Use `./configure -help` to get a list of the available arguments. Here are some of the ones you might need:

`-with-ghc=`*`path`*

Specifies the path to an installed GHC which you would like to use. This compiler will be used for compiling GHC-specific code (eg. GHC itself). This option *cannot* be specified using `build.mk` (see later), because **configure** needs to auto-detect the version of GHC you're using. The default is to look for a compiler named `ghc` in your path.

`-with-hc=`*`path`*

Specifies the path to any installed Haskell compiler. This compiler will be used for compiling generic Haskell code. The default is to use `ghc`.

configure caches the results of its run in `config.cache`. Quite often you don't want that; you're running **configure** a second time because something has changed. In that case, simply delete `config.cache`.

Step 3: build configuration.

Next, you say how this build of `fptools` is to differ from the standard defaults by creating a new file `mk/build.mk` *in the build tree*. This file is the one and only file you edit in the build tree, precisely because it says how this build differs from the source. (Just in case your build tree does die, you might want to keep a private directory of `build.mk` files, and use a symbolic link in each build tree to point to the appropriate one.) So `mk/build.mk` never exists in the source tree—you create one in each build tree from the template. We'll discuss what to put in it shortly.

And that's it for configuration. Simple, eh?

What do you put in your build-specific configuration file `mk/build.mk`? *For almost all purposes all you will do is put make variable definitions that override those in* `mk/config.mk.in`. The whole point of `mk/config.mk.in`—and its derived counterpart `mk/config.mk`—is to define the build configuration. It is heavily commented, as you will see if you look at it. So generally, what you do is look at `mk/config.mk.in`, and add definitions in `mk/build.mk` that override any of the `config.mk` definitions that you want to change. (The override occurs because the main boilerplate file, `mk/boilerplate.mk`, includes `build.mk` after `config.mk`.)

For example, `config.mk.in` contains the definition:

```
GhcHcOpts=-O -Rghc-timing
```

The accompanying comment explains that this is the list of flags passed to GHC when building GHC itself. For doing development, it is wise to add `-DDEBUG`, to enable debugging code. So you would add the following to `build.mk`:

or, if you prefer,

```
GhcHcOpts += -DDEBUG
```

GNU **make** allows existing definitions to have new text appended using the “`+=`” operator, which is quite a convenient feature.)

If you want to remove the `-O` as well (a good idea when developing, because the turn-around cycle gets a lot quicker), you can just override `GhcLibHcOpts` altogether:

```
GhcHcOpts=-DDEBUG -Rghc-timing
```

When reading `config.mk.in`, remember that anything between “@...@” signs is going to be substituted by **configure** later. You *can* override the resulting definition if you want, but you need to be a bit surer what you are doing. For example, there’s a line that says:

```
YACC = @YaccCmd@
```

This defines the Make variables `YACC` to the pathname for a **yacc** that **configure** finds somewhere. If you have your own pet **yacc** you want to use instead, that’s fine. Just add this line to `mk/build.mk`:

```
YACC = myyacc
```

You do not *have* to have a `mk/build.mk` file at all; if you don’t, you’ll get all the default settings from `mk/config.mk.in`.

You can also use `build.mk` to override anything that **configure** got wrong. One place where this happens often is with the definition of `FPTOOLS_TOP_ABS`: this variable is supposed to be the canonical path to the top of your source tree, but if your system uses an automounter then the correct directory is hard to find automatically. If you find that **configure** has got it wrong, just put the correct definition in `build.mk`.

5.4. The story so far

Let’s summarise the steps you need to carry to get yourself a fully-configured build tree from scratch.

1. Get your source tree from somewhere (CVS repository or source distribution). Say you call the root directory `myfptools` (it does not have to be called `fptools`). Make sure that you have the essential files (see Section 5.1).
2. Use **lndir** or **mkshadowdir** to create a build tree.

   ```
   cd myfptools
   mkshadowdir . /scratch/joe-bloggs/myfptools-sun4
   ```

(N.B. **mkshadowdir**’s first argument is taken relative to its second.) You probably want to give the build tree a name that suggests its main defining characteristic (in your mind at least), in case you later add others.

3. Change directory to the build tree. Everything is going to happen there now.

   ```
   cd /scratch/joe-bloggs/myfptools-sun4
   ```

4. Prepare for system configuration:

   ```
   autoconf
   ```

(You can skip this step if you are starting from a source distribution, and you already have `configure` and `mk/config.h.in`.)

5. Do system configuration:

   ```
   ./configure
   ```

6. Create the file `mk/build.mk`, adding definitions for your desired configuration options.

   ```
   emacs mk/build.mk
   ```

You can make subsequent changes to `mk/build.mk` as often as you like. You do not have to run any further configuration programs to make these changes take effect. In theory you should, however, say **gmake clean**, **gmake all**, because configuration option changes could affect anything—but in practice you are likely to know what's affected.

5.5. Making things

At this point you have made yourself a fully-configured build tree, so you are ready to start building real things.

The first thing you need to know is that *you must use GNU **make**, usually called **gmake**, not standard Unix **make***. If you use standard Unix **make** you will get all sorts of error messages (but no damage) because the `fptools` **Makefiles** use GNU **make**'s facilities extensively.

To just build the whole thing, **cd** to the top of your `fptools` tree and type **gmake**. This will prepare the tree and build the various projects in the correct order.

5.6. Standard Targets

In any directory you should be able to make the following:

`boot`:

: does the one-off preparation required to get ready for the real work. Notably, it does **gmake depend** in all directories that contain programs. It also builds the necessary tools for compilation to proceed.

  Invoking the `boot` target explicitly is not normally necessary. From the top-level `fptools` directory, invoking `gmake` causes `gmake boot all` to be invoked in each of the project subdirectories, in the order specified by `$(AllTargets)` in `config.mk`.

  If you're working in a subdirectory somewhere and need to update the dependencies, `gmake boot` is a good way to do it.

`all`:

makes all the final target(s) for this Makefile. Depending on which directory you are in a “final target” may be an executable program, a library archive, a shell script, or a Postscript file. Typing **gmake** alone is generally the same as typing **gmake all**.

`install`:

installs the things built by `all`. Where does it install them? That is specified by `mk/config.mk.in`; you can override it in `mk/build.mk`, or by running **configure** with command-line arguments like `-bindir=/home/simonpj/bin`; see `./configure -help` for the full details.

`uninstall`:

reverses the effect of `install`.

`clean`:

Delete all files from the current directory that are normally created by building the program. Don’t delete the files that record the configuration, or files generated by **gmake boot**. Also preserve files that could be made by building, but normally aren’t because the distribution comes with them.

`distclean`:

Delete all files from the current directory that are created by configuring or building the program. If you have unpacked the source and built the program without creating any other files, `make distclean` should leave only the files that were in the distribution.

`mostlyclean`:

Like `clean`, but may refrain from deleting a few files that people normally don’t want to recompile.

`maintainer-clean`:

Delete everything from the current directory that can be reconstructed with this Makefile. This typically includes everything deleted by `distclean`, plus more: C source files produced by Bison, tags tables, Info files, and so on.

One exception, however: `make maintainer-clean` should not delete `configure` even if `configure` can be remade using a rule in the `Makefile`. More generally, `make maintainer-clean` should not delete anything that needs to exist in order to run `configure` and then begin to build the program.

`check`:

run the test suite.

All of these standard targets automatically recurse into sub-directories. Certain other standard targets do not:

`configure`:

is only available in the root directory `$(FPTOOLS_TOP)`; it has been discussed in Section 5.3.

`depend`:

make a `.depend` file in each directory that needs it. This `.depend` file contains mechanically-generated dependency information; for example, suppose a directory contains a Haskell source module `Foo.lhs` which imports another module `Baz`. Then the generated `.depend` file will contain the dependency:

```
Foo.o : Baz.hi
```

which says that the object file `Foo.o` depends on the interface file `Baz.hi` generated by compiling module `Baz`. The `.depend` file is automatically included by every Makefile.

`binary-dist`:

make a binary distribution. This is the target we use to build the binary distributions of GHC and Happy.

`dist`:

make a source distribution. Note that this target does “make distclean” as part of its work; don’t use it if you want to keep what you’ve built.

Most `Makefile`s have targets other than these. You can discover them by looking in the `Makefile` itself.

5.7. Using a project from the build tree

If you want to build GHC (say) and just use it direct from the build tree without doing `make install` first, you can run the in-place driver script: `ghc/compiler/ghc-inplace`.

Do *NOT* use `ghc/compiler/ghc`, or `ghc/compiler/ghc-5.xx`, as these are the scripts intended for installation, and contain hard-wired paths to the installed libraries, rather than the libraries in the build tree.

Happy can similarly be run from the build tree, using `happy/src/happy-inplace`.

5.8. Fast Making

Sometimes the dependencies get in the way: if you’ve made a small change to one file, and you’re absolutely sure that it won’t affect anything else, but you know that **make** is going to rebuild everything anyway, the following hack may be useful:

```
gmake FAST=YES
```

This tells the make system to ignore dependencies and just build what you tell it to. In other words, it's equivalent to temporarily removing the `.depend` file in the current directory (where **mkdependHS** and friends store their dependency information).

A bit of history: GHC used to come with a **fastmake** script that did the above job, but GNU make provides the features we need to do it without resorting to a script. Also, we've found that fastmaking is less useful since the advent of GHC's recompilation checker (see the User's Guide section on "Separate Compilation").

6. The `Makefile` architecture

make is great if everything works—you type **gmake install** and lo! the right things get compiled and installed in the right places. Our goal is to make this happen often, but somehow it often doesn't; instead some weird error message eventually emerges from the bowels of a directory you didn't know existed.

The purpose of this section is to give you a road-map to help you figure out what is going right and what is going wrong.

6.1. A small project

To get started, let us look at the `Makefile` for an imaginary small `fptools` project, `small`. Each project in `fptools` has its own directory in `FPTOOLS_TOP`, so the `small` project will have its own directory `FPOOLS_TOP/small/`. Inside the `small/` directory there will be a `Makefile`, looking something like this:

```
#     Makefile for fptools project "small"

TOP = ..
include $(TOP)/mk/boilerplate.mk

SRCS = $(wildcard *.lhs) $(wildcard *.c)
HS_PROG = small

include $(TOP)/target.mk
```

This `Makefile` has three sections:

1. The first section includes [1] a file of “boilerplate” code from the level above (which in this case will be `FPTOOLS_TOP/mk/boilerplate.mk`). As its name suggests, `boilerplate.mk` consists of a large quantity of standard `Makefile` code. We discuss this boilerplate in more detail in Section 6.4. Before the `include` statement, you must define the **make** variable `TOP` to be the directory containing the `mk` directory in which the `boilerplate.mk` file is. It is *not* OK to simply say

   ```
   include ../mk/boilerplate.mk  # NO NO NO
   ```

Why? Because the `boilerplate.mk` file needs to know where it is, so that it can, in turn, `include` other files. (Unfortunately, when an `include`d file does an `include`, the filename is treated relative to the directory in which **gmake** is being run, not the directory in which the `include`d sits.) In general, *every file `foo.mk` assumes that `$(TOP)/mk/foo.mk` refers to itself.* It is up to the `Makefile` doing the `include` to ensure this is the case. Files intended for inclusion in other `Makefile`s are written to have the following property: *after `foo.mk` is `include`d, it leaves `TOP` containing the same value as it had just before the `include` statement*. In our example, this invariant guarantees that the `include` for `target.mk` will look in the same directory as that for `boilerplate.mk`.

2. The second section defines the following standard **make** variables: `SRCS` (the source files from which is to be built), and `HS_PROG` (the executable binary to be built). We will discuss in more detail what the “standard variables” are, and how they affect what happens, in Section 6.6. The definition for `SRCS` uses the useful GNU **make** construct `$(wildcard $pat$)`, which expands to a list of all the files matching the pattern `pat` in the current directory. In this example, `SRCS` is set to the list of all the `.lhs` and `.c` files in the directory. (Let’s suppose there is one of each, `Foo.lhs` and `Baz.c`.)

3. The last section includes a second file of standard code, called `target.mk`. It contains the rules that tell **gmake** how to make the standard targets (Section 5.6). Why, you ask, can’t this standard code be part of `boilerplate.mk`? Good question. We discuss the reason later, in Section 6.3. You do not *have* to `include` the `target.mk` file. Instead, you can write rules of your own for all the standard targets. Usually, though, you will find quite a big payoff from using the canned rules in `target.mk`; the price tag is that you have to understand what canned rules get enabled, and what they do (Section 6.6).

In our example `Makefile`, most of the work is done by the two `include`d files. When you say **gmake all**, the following things happen:

- **gmake** figures out that the object files are `Foo.o` and `Baz.o`.
- It uses a boilerplate pattern rule to compile `Foo.lhs` to `Foo.o` using a Haskell compiler. (Which one? That is set in the build configuration.)
- It uses another standard pattern rule to compile `Baz.c` to `Baz.o`, using a C compiler. (Ditto.)
- It links the resulting `.o` files together to make `small`, using the Haskell compiler to do the link step. (Why not use **ld**? Because the Haskell compiler knows what standard libraries to link in.

How did **gmake** know to use the Haskell compiler to do the link, rather than the C compiler? Because we set the variable `HS_PROG` rather than `C_PROG`.)

All `Makefile`s should follow the above three-section format.

6.2. A larger project

Larger projects are usually structured into a number of sub-directories, each of which has its own `Makefile`. (In very large projects, this sub-structure might be iterated recursively, though that is rare.) To give you the idea, here's part of the directory structure for the (rather large) GHC project:

```
$(FPTOOLS_TOP)/ghc/
  Makefile
  mk/
    boilerplate.mk
    rules.mk
   docs/
    Makefile
    ...source files for documentation...
   driver/
    Makefile
    ...source files for driver...
   compiler/
    Makefile
    parser/...source files for parser...
    renamer/...source files for renamer...
    ...etc...
```

The sub-directories `docs`, `driver`, `compiler`, and so on, each contains a sub-component of GHC, and each has its own `Makefile`. There must also be a `Makefile` in `$(FPTOOLS_TOP)/ghc`. It does most of its work by recursively invoking **gmake** on the `Makefile`s in the sub-directories. We say that `ghc/Makefile` is a *non-leaf `Makefile`*, because it does little except organise its children, while the `Makefile`s in the sub-directories are all *leaf `Makefile`s*. (In principle the sub-directories might themselves contain a non-leaf `Makefile` and several sub-sub-directories, but that does not happen in GHC.)

The `Makefile` in `ghc/compiler` is considered a leaf `Makefile` even though the `ghc/compiler` has sub-directories, because these sub-directories do not themselves have `Makefile`s in them. They are just used to structure the collection of modules that make up GHC, but all are managed by the single `Makefile` in `ghc/compiler`.

You will notice that `ghc/` also contains a directory `ghc/mk/`. It contains GHC-specific `Makefile` boilerplate code. More precisely:

- `ghc/mk/boilerplate.mk` is included at the top of `ghc/Makefile`, and of all the leaf `Makefiles` in the sub-directories. It in turn `includes` the main boilerplate file `mk/boilerplate.mk`.
- `ghc/mk/target.mk` is `included` at the bottom of `ghc/Makefile`, and of all the leaf `Makefiles` in the sub-directories. It in turn `includes` the file `mk/target.mk`.

So these two files are the place to look for GHC-wide customisation of the standard boilerplate.

6.3. Boilerplate architecture

Every `Makefile` includes a `boilerplate.mk` file at the top, and `target.mk` file at the bottom. In this section we discuss what is in these files, and why there have to be two of them. In general:

- `boilerplate.mk` consists of:
  - *Definitions of millions of **make** variables* that collectively specify the build configuration. Examples: `HC_OPTS`, the options to feed to the Haskell compiler; `NoFibSubDirs`, the sub-directories to enable within the `nofib` project; `GhcWithHc`, the name of the Haskell compiler to use when compiling GHC in the `ghc` project.
  - *Standard pattern rules* that tell **gmake** how to construct one file from another.

  `boilerplate.mk` needs to be `included` at the *top* of each `Makefile`, so that the user can replace the boilerplate definitions or pattern rules by simply giving a new definition or pattern rule in the `Makefile`. **gmake** simply takes the last definition as the definitive one. Instead of *replacing* boilerplate definitions, it is also quite common to *augment* them. For example, a `Makefile` might say:

  ```
  SRC_HC_OPTS += -O
  ```

  thereby adding “`-O`” to the end of `SRC_HC_OPTS`.
- `target.mk` contains **make** rules for the standard targets described in Section 5.6. These rules are selectively included, depending on the setting of certain **make** variables. These variables are usually set in the middle section of the `Makefile` between the two `includes`. `target.mk` must be included at the end (rather than being part of `boilerplate.mk`) for several tiresome reasons:
  - **gmake** commits target and dependency lists earlier than it should. For example, `target.mk` has a rule that looks like this:

    ```
    $(HS_PROG) : $(OBJS)
          $(HC) $(LD_OPTS) $< -o $@
    ```

    If this rule was in `boilerplate.mk` then `$(HS_PROG)` and `$(OBJS)` would not have their final values at the moment **gmake** encountered the rule. Alas, **gmake** takes a snapshot of their current values, and wires that snapshot into the rule. (In contrast, the commands executed when the rule “fires” are only substituted at the moment of firing.) So, the rule must follow the definitions given in the `Makefile` itself.

- Unlike pattern rules, ordinary rules cannot be overriden or replaced by subsequent rules for the same target (at least, not without an error message). Including ordinary rules in `boilerplate.mk` would prevent the user from writing rules for specific targets in specific cases.
- There are a couple of other reasons I've forgotten, but it doesn't matter too much.

6.4. The main `mk/boilerplate.mk` file

If you look at `$(FPTOOLS_TOP)/mk/boilerplate.mk` you will find that it consists of the following sections, each held in a separate file:

`config.mk`

is the build configuration file we discussed at length in Section 5.3.

`paths.mk`

defines **make** variables for pathnames and file lists. In particular, it gives definitions for:

`SRCS`:

all source files in the current directory.

`HS_SRCS`:

all Haskell source files in the current directory. It is derived from `$(SRCS)`, so if you override `SRCS` with a new value `HS_SRCS` will follow suit.

`C_SRCS`:

similarly for C source files.

`HS_OBJS`:

the `.o` files derived from `$(HS_SRCS)`.

`C_OBJS`:

similarly for `$(C_SRCS)`.

`OBJS`:

the concatenation of `$(HS_OBJS)` and `$(C_OBJS)`.

Any or all of these definitions can easily be overriden by giving new definitions in your `Makefile`. For example, if there are things in the current directory that look like source files but aren't, then you'll need to set `SRCS` manually in your `Makefile`. The other definitions will then work from this new definition.

What, exactly, does `paths.mk` consider a “source file” to be? It’s based on the file’s suffix (e.g. `.hs`, `.lhs`, `.c`, `.lc`, etc), but this is the kind of detail that changes, so rather than enumerate the source suffices here the best thing to do is to look in `paths.mk`.

`opts.mk`

defines **make** variables for option strings to pass to each program. For example, it defines `HC_OPTS`, the option strings to pass to the Haskell compiler. See Section 6.5.

`suffix.mk`

defines standard pattern rules—see Section 6.5.

Any of the variables and pattern rules defined by the boilerplate file can easily be overridden in any particular `Makefile`, because the boilerplate `include` comes first. Definitions after this `include` directive simply override the default ones in `boilerplate.mk`.

6.5. Pattern rules and options

The file `suffix.mk` defines standard *pattern rules* that say how to build one kind of file from another, for example, how to build a `.o` file from a `.c` file. (GNU **make**’s *pattern rules* are more powerful and easier to use than Unix **make**’s *suffix rules*.)

Almost all the rules look something like this:

```
%.o : %.c
      $(RM) $@
      $(CC) $(CC_OPTS) -c $< -o $@
```

Here’s how to understand the rule. It says that *something*`.o` (say `Foo.o`) can be built from *something*`.c` (`Foo.c`), by invoking the C compiler (path name held in `$(CC)`), passing to it the options `$(CC_OPTS)` and the rule’s dependent file of the rule `$<` (`Foo.c` in this case), and putting the result in the rule’s target `$@` (`Foo.o` in this case).

Every program is held in a **make** variable defined in `mk/config.mk`—look in `mk/config.mk` for the complete list. One important one is the Haskell compiler, which is called `$(HC)`.

Every program’s options are are held in a **make** variables called `<prog>_OPTS`. the `<prog>_OPTS` variables are defined in `mk/opts.mk`. Almost all of them are defined like this:

```
CC_OPTS = $(SRC_CC_OPTS) $(WAY$(_way)_CC_OPTS) $($*_CC_OPTS) $(EX-
TRA_CC_OPTS)
```

The four variables from which `CC_OPTS` is built have the following meaning:

`SRC_CC_OPTS`:

options passed to all C compilations.

`WAY_<way>_CC_OPTS`:

options passed to C compilations for way `<way>`. For example, `WAY_mp_CC_OPTS` gives options to pass to the C compiler when compiling way `mp`. The variable `WAY_CC_OPTS` holds options to pass to the C compiler when compiling the standard way. (Section 6.8 dicusses multi-way compilation.)

`<module>_CC_OPTS`:

options to pass to the C compiler that are specific to module `<module>`. For example, `SMap_CC_OPTS` gives the specific options to pass to the C compiler when compiling `SMap.c`.

`EXTRA_CC_OPTS`:

extra options to pass to all C compilations. This is intended for command line use, thus:

```
gmake libHS.a EXTRA_CC_OPTS="-v"
```

6.6. The main `mk/target.mk` file

`target.mk` contains canned rules for all the standard targets described in Section 5.6. It is complicated by the fact that you don't want all of these rules to be active in every `Makefile`. Rather than have a plethora of tiny files which you can include selectively, there is a single file, `target.mk`, which selectively includes rules based on whether you have defined certain variables in your `Makefile`. This section explains what rules you get, what variables control them, and what the rules do. Hopefully, you will also get enough of an idea of what is supposed to happen that you can read and understand any weird special cases yourself.

`HS_PROG`.

If `HS_PROG` is defined, you get rules with the following targets:

`HS_PROG`

itself. This rule links `$(OBJS)` with the Haskell runtime system to get an executable called `$(HS_PROG)`.

`install`

installs `$(HS_PROG)` in `$(bindir)`.

`C_PROG`

is similar to `HS_PROG`, except that the link step links `$(C_OBJS)` with the C runtime system.

LIBRARY

is similar to HS_PROG, except that it links $(LIB_OBJS) to make the library archive $(LIBRARY), and install installs it in $(libdir).

LIB_DATA

...

LIB_EXEC

...

HS_SRCS, C_SRCS.

If HS_SRCS is defined and non-empty, a rule for the target depend is included, which generates dependency information for Haskell programs. Similarly for C_SRCS.

All of these rules are "double-colon" rules, thus

```
install :: $(HS_PROG)
      ...how to install it...
```

GNU **make** treats double-colon rules as separate entities. If there are several double-colon rules for the same target it takes each in turn and fires it if its dependencies say to do so. This means that you can, for example, define both HS_PROG and LIBRARY, which will generate two rules for install. When you type **gmake install** both rules will be fired, and both the program and the library will be installed, just as you wanted.

6.7. Recursion

In leaf Makefiles the variable SUBDIRS is undefined. In non-leaf Makefiles, SUBDIRS is set to the list of sub-directories that contain subordinate Makefiles. *It is up to you to set SUBDIRS in the Makefile.* There is no automation here—SUBDIRS is too important to automate.

When SUBDIRS is defined, target.mk includes a rather neat rule for the standard targets (Section 5.6 that simply invokes **make** recursively in each of the sub-directories.

These recursive invocations are guaranteed to occur in the order in which the list of directories is specified in SUBDIRS. This guarantee can be important. For example, when you say **gmake boot** it can be important that the recursive invocation of **make boot** is done in one sub-directory (the include files, say) before another (the source files). Generally, put the most independent sub-directory first, and the most dependent last.

6.8. Way management

We sometimes want to build essentially the same system in several different "ways". For example, we want to build GHC's Prelude libraries with and without profiling, with and without concurrency, and so on, so that there is an appropriately-built library archive to link with when the user compiles his program. It would be possible to have a completely separate build tree for each such "way", but it would be horribly bureaucratic, especially since often only parts of the build tree need to be constructed in multiple ways.

Instead, the `target.mk` contains some clever magic to allow you to build several versions of a system; and to control locally how many versions are built and how they differ. This section explains the magic.

The files for a particular way are distinguished by munging the suffix. The "normal way" is always built, and its files have the standard suffices `.o`, `.hi`, and so on. In addition, you can build one or more extra ways, each distinguished by a *way tag*. The object files and interface files for one of these extra ways are distinguished by their suffix. For example, way `mp` has files `.mp_o` and `.mp_hi`. Library archives have their way tag the other side of the dot, for boring reasons; thus, `libHS_mp.a`.

A **make** variable called `way` holds the current way tag. *`way` is only ever set on the command line of a recursive invocation of **gmake***. It is never set inside a `Makefile`. So it is a global constant for any one invocation of **gmake**. Two other **make** variables, `way_` and `_way` are immediately derived from `$(way)` and never altered. If `way` is not set, then neither are `way_` and `_way`, and the invocation of **make** will build the "normal way". If `way` is set, then the other two variables are set in sympathy. For example, if `$(way)` is "`mp`", then `way_` is set to "`mp_`" and `_way` is set to "`_mp`". These three variables are then used when constructing file names.

So how does **make** ever get recursively invoked with `way` set? There are two ways in which this happens:

- For some (but not all) of the standard targets, when in a leaf sub-directory, **make** is recursively invoked for each way tag in `$(WAYS)`. You set `WAYS` to the list of way tags you want these targets built for. The mechanism here is very much like the recursive invocation of **make** in sub-directories (Section 6.7). It is up to you to set `WAYS` in your `Makefile`; this is how you control what ways will get built.
- For a useful collection of targets (such as `libHS_mp.a`, `Foo.mp_o`) there is a rule which recursively invokes **make** to make the specified target, setting the `way` variable. So if you say **gmake Foo.mp_o** you should see a recursive invocation **gmake Foo.mp_o way=mp**, and *in this recursive invocation the pattern rule for compiling a Haskell file into a `.o` file will match*. The key pattern rules (in `suffix.mk`) look like this:

  ```
  %.$(way_)o : %.lhs
          $(HC) $(HC_OPTS) $< -o $@
  ```

Neat, eh?

6.9. When the canned rule isn't right

Sometimes the canned rule just doesn't do the right thing. For example, in the `nofib` suite we want the link step to print out timing information. The thing to do here is *not* to define `HS_PROG` or

`C_PROG`, and instead define a special purpose rule in your own `Makefile`. By using different variable names you will avoid the canned rules being included, and conflicting with yours.

7. Booting/porting from C (`.hc`) files

This section is for people trying to get GHC going by using the supplied intermediate C (`.hc`) files. This would probably be because no binaries have been provided, or because the machine is not “fully supported”.

The intermediate C files are normally made available together with a source release, please check the announce message for exact directions of where to find them. If we haven’t made them available or you can’t find them, please ask.

Assuming you’ve got them, unpack them on top of a fresh source tree. This will place matching `.hc` files next to the corresponding Haskell source in the compiler subdirectory `ghc` and in the language package of hslibs (i.e., in `hslibs/lang`). Then follow the ‘normal’ instructions in Section 5 for setting up a build tree.

The actual build process is fully automated by the `hc-build` script located in the `distrib` directory. If you eventually want to install GHC into the directory `INSTALL_DIRECTORY`, the following command will execute the whole build process (it won’t install yet):

```
foo% distrib/hc-build -prefix=INSTALL_DIRECTORY
```

By default, the installation directory is `/usr/local`. If that is what you want, you may omit the argument to `hc-build`. Generally, any option given to `hc-build` is passed through to the configuration script `configure`. If `hc-build` successfully completes the build process, you can install the resulting system, as normal, with

```
foo% make install
```

That’s the mechanics of the boot process, but, of course, if you’re trying to boot on a platform that is not supported and significantly ‘different’ from any of the supported ones, this is only the start of the adventure. . . (ToDo: porting tips—stuff to look out for, etc.)

8. Known pitfalls in building Glasgow Haskell

WARNINGS about pitfalls and known “problems”:

1. One difficulty that comes up from time to time is running out of space in `TMPDIR`. (It is impossible for the configuration stuff to compensate for the vagaries of different sysadmin approaches to temp space.) The quickest way around it is **setenv TMPDIR /usr/tmp** or even **setenv TMPDIR .** (or the equivalent incantation with your shell of choice). The best way around it is to say

```
export TMPDIR=<dir>
```

in your `build.mk` file. Then GHC and the other `fptools` programs will use the appropriate directory in all cases.

2. In compiling some support-code bits, e.g., in `ghc/rts/gmp` and even in `ghc/lib`, you may get a few C-compiler warnings. We think these are OK.
3. When compiling via C, you'll sometimes get "warning: assignment from incompatible pointer type" out of GCC. Harmless.
4. Similarly, **ar**chiving warning messages like the following are not a problem:

```
ar: filename GlaIOMonad__1_2s.o truncated to GlaIOMonad_
ar: filename GlaIOMonad__2_2s.o truncated to GlaIOMonad_
...
```

5. In compiling the compiler proper (in `compiler/`), you *may* get an "Out of heap space" error message. These can vary with the vagaries of different systems, it seems. The solution is simple:
   - If you're compiling with GHC 4.00 or later, then the *maximum* heap size must have been reached. This is somewhat unlikely, since the maximum is set to 64M by default. Anyway, you can raise it with the `-optCrts-M<size>` flag (add this flag to `<module>_HC_OPTS` **make** variable in the appropriate `Makefile`).
   - For GHC < 4.00, add a suitable `-H` flag to the `Makefile`, as above.

and try again: **gmake**. (see Section 6.5 for information about `<module>_HC_OPTS`.) Alternatively, just cut to the chase:

```
% cd ghc/compiler
% make EXTRA_HC_OPTS=-optCrts-M128M
```

6. If you try to compile some Haskell, and you get errors from GCC about lots of things from `/usr/include/math.h`, then your GCC was mis-installed. **fixincludes** wasn't run when it should've been. As **fixincludes** is now automagically run as part of GCC installation, this bug also suggests that you have an old GCC.
7. You *may* need to re-**ranlib** your libraries (on Sun4s).

```
% cd $(libdir)/ghc-x.xx/sparc-sun-sunos4
% foreach i ( ‘find . -name ’*.a’ -print‘ ) # or other-shell equiv...
?    ranlib $i
?    # or, on some machines: ar s $i
? end
```

We'd be interested to know if this is still necessary.

8. GHC's sources go through **cpp** before being compiled, and **cpp** varies a bit from one Unix to another. One particular gotcha is macro calls like this:

```
SLIT("Hello, world")
```

Some **cpp**s treat the comma inside the string as separating two macro arguments, so you get

```
:731: macro ‘SLIT’ used with too many (2) args
```

Alas, **cpp** doesn't tell you the offending file! Workaround: don't put weird things in string args to **cpp** macros.

9. Notes for building under Windows

This section summarises how to get the utilities you need on your Win95/98/NT/2000 machine to use CVS and build GHC. Similar notes for installing and running GHC may be found in the user guide. In general, Win95/Win98 behave the same, and WinNT/Win2k behave the same. You should read the GHC installation guide sections on Windows (in the user guide) before continuing to read these notes.

Because of various hard-wired infelicities, you need to copy `bash.exe` (from GHC's `extra-bin` directory), and `perl.exe` and `cat.exe` (from GHC's `bin` directory) to `/bin` (discover where your Cygwin root directory is by typign **mount**). You also need to have `extra-bin` on your path.

Before you start, you need to make sure that the user environment variable `MAKE_MODE` is set to `UNIX`. If you don't do this you get very weird messages when you type **make**, such as:

```
/c: /c: No such file or directory
```

9.1. Configuring ssh

- Generate a key, by running `c:/user/local/bin/ssh-keygen1`. This generates a public key in `.ssh/identity.pub`, and a private key in `.ssh/identity`

  In response to the 'Enter passphrase' question, just hit return (i.e. use an empty passphrase). The passphrase is a password that protects your private key. But it's a pain to type this passphrase everytime you use **ssh**, so the best thing to do is simply to protect your `.ssh` directory, and `.ssh/identity` from access by anyone else. To do this right-click your `.ssh` directory, and select Properties. If you are not on the access control list, add yourself, and give yourself full permissions (the second panel). Remove everyone else from the access control list. (Don't leave them there but deny them access, because 'they' may be a list that includes you!)

  If you have problems running **ssh-keygen1** from within **bash**, start up `cmd.exe` and run it as follows:

  ```
  c:\tmp> set CYGWIN32=tty
  c:\tmp> c:/user/local/bin/ssh-keygen1
  ```

- If you don't have an account on `cvs.haskell.org`, send your `.ssh/identity.pub` to the CVS repository administrator (currently Jeff Lewis <`jlewis@cse.ogi.edu`>). He will set up your account.

  If you do have an account on `cvs.haskell.org`, use TeraTerm to logon to it. Once in, copy the key that **ssh-keygen1** deposited in `/.ssh/identity.pub` into your

`~/.ssh/authorized_keys`. Make sure that the new version of `authorized_keys` still has 600 file permission.

9.2. Configuring CVS

- From the System control panel, set the following *user* environment variables (see the GHC user guide)
  - `HOME`: points to your home directory. This is where CVS will look for its `.cvsrc` file.
  - `CVS_RSH`: `c:/path_to_ghc/extra-bin/ssh`
  - `CVSROOT`: `:ext:username@cvs.haskell.org:/home/cvs/root`, where `username` is your userid
  - `CVSEDITOR`: `bin/gnuclient.exe` if you want to use an Emacs buffer for typing in those long commit messages.
  - `SHELL`: To use bash as the shell in Emacs, you need to set this to point to `bash.exe`.
- Put the following in `$HOME/.cvsrc`:

  ```
  checkout -P
  release -d
  update -P
  diff -u
  ```

  These are the default options for the specified CVS commands, and represent better defaults than the usual ones. (Feel free to change them.)

  Filenames starting with `.` were illegal in the 8.3 DOS filesystem, but that restriction should have been lifted by now (i.e., you're using VFAT or later filesystems.) If you're still having problems creating it, don't worry; `.cvsrc` is entirely optional.
- Try doing **cvs co fpconfig**. All being well, bytes should start to trickle through, leaving a directory `fptools` in your current directory. (You can **rm** it if you don't want to keep it.) The following messages appear to be harmless:

  ```
  setsockopt IPTOS_LOWDELAY: Invalid argument
  setsockopt IPTOS_THROUGHPUT: Invalid argument
  ```

  At this point I found that CVS tried to invoke a little dialogue with me (along the lines of 'do you want to talk to this host?'), but for some reason bombed out. This was from a bash shell running in Emacs. I solved this by invoking a Cygnus shell, and running CVS from there. Once things are dialogue free, it seems to work OK from within Emacs.
- If you want to check out part of large tree, proceed as follows:

  ```
  cvs -f checkout -l papers
  cd papers
  cvs update cpr
  ```

This sequence checks out the `papers` module, but none of its sub-directories. The "`-l`" flag says not to check out sub-directories. The "`-f`" flag says not to read the `.cvsrc` file whose `-P` default (don't check out empty directories) is in this case bogus.

The **cvs update** command sucks in a named sub-directory.

There is a very nice graphical front-end to CVS for Win32 platforms, with a UI that people will be familiar with, at wincvs.org (http://www.wincvs.org/). I have not tried it yet.

9.3. Building GHC

- In the `./configure` output, ignore "`checking whether #! works in shell scripts... ./configure: ./conftest: No such file or directory`", and "`not updating unwritable cache ./config.cache`". Nobody knows why these happen, but they seem to be harmless.
- You have to run **autoconf** both in `fptools` and in `fptools/ghc`. If you omit the latter step you'll get an error when you run `./configure`:

```
...lots of stuff...
creating mk/config.h
mk/config.h is unchanged
configuring in ghc
running /bin/sh ./configure  -cache-file=.././config.cache -srcdir=.
./configure: ./configure: No such file or directory
configure: error: ./configure failed for ghc
```

- You need `ghc` to be in your `PATH` before you run **configure**. The default GHC InstallShield creates only `ghc-4.08`, so you may need to duplicate this file as `ghc` in the same directory, in order that **configure** will see it (or just rename `ghc-4.08` to `ghc`. And make sure that the directory is in your path.

Notes

1. One of the most important features of GNU **make** that we use is the ability for a `Makefile` to include another named file, very like **cpp**'s `#include` directive.