The Haskell Cabal

A Common Architecture for Building Applications and Tools

Isaac Jones

Simon Peyton Jones

Simon Marlow

Malcolm Wallace

Ross Patterson

The Haskell Library and Tools Infrastructure Project is an effort to provide a framework for developers to more effectively contribute their software to the Haskell community. This document specifies the *Common Architecture for Building Applications and Tools(Cabal)*, which contributes to the goals of the Haskell Library and Tools Infrastructure Project.

Specifically, the Cabal describes what a Haskell package is, how these packages interact with the language, and what Haskell implementations must to do to support packages. The Cabal also specifies some infrastructure (code) that makes it easy for tool authors to build and distribute conforming packages.

The Cabal is only one contribution to the Library Infrastructure project. In particular, the Cabal says nothing about more global issues such as how authors decide where in the module name space their library should live; how users can find a package they want; how orphan packages find new owners; and so on.

The Cabal has been discussed by the implementors of GHC, Nhc98, and Hugs, all of whom are prepared to implement it. The proposal is now open for wider debate. Please contribute by emailing <`libraries@haskell.org`>.

1. The Haskell Package System: goals

The Haskell Package System (Cabal) has the following main goal: to specify a standard way in which a Haskell tool can be packaged, so that it is easy for consumers to use it, or re-package it, regardless of the Haskell implementation or installation platform.

The Cabal also supports tool authors by providing an infrastructure that automates the process of building and packaging simple tools. It is not necessary to use this code—indeed complex libraries may exceed its abilities—but it should handle many cases with no trouble.

1.1. Dramatis personae

The Cabal serves a number of people in different ways:

- *Joe User* is simply a Haskell user. He does not download new packages. Nevertheless, he needs to know about his Haskell compiler's `-package` flag (see Section 3).
- *Bob the Builder* and *Sam Sysadmin* both download, build, and install new packages. The only difference between the two is that Sam has root permission, and can install packages in more globally-visible places.
- *Peter Packager* builds operating system specific install files (e.g. .msi .rpm .deb) from packages supplied by Marcus or Angela. We might also call him *Roland RPM*, *Donald Debian*, and *Willie Windows* who build Linux RPM, Debian, and Windows installer packages respectively (this list is not exhaustive). They do this as a service to their platform's community, and may know little or nothing about the internal details of the Haskell packages they are wrapping up.
- *Isabella Installer* installs binary packages supplied by Peter or Angela, (or Rowland, Donald, and Willie). Isabella requires only a Haskell compiler/interpreter. She can use rpm to install packages by Rowland. She cannot or will not build the packages herself, so she relies on Peter to provide them. She won't use the Setup script directly from Angela, but she might use a layered tool like haskell-install, which does all the work of downloading and installing simple packages.
- *Angela Author* wants to write a simple Haskell tool, and distribute it with minimum fuss, in such a way that all the above folk can easily use it.
- *Marcus Makefile* is like Angela, but more sophisticated. He has a complicated tool, and uses makefiles. Still, he wants to arrange that Roland, Donald, Bob, Sam, and Joe don't need to know about his internal complexity.

We describe Angela and Marcus as *producers* of their packages, and all the others as package *consumers*.

Note that though these users all have different names, it is very common for their roles to overlap when it comes to real people. For instance, if Bob builds packages for himself, he becomes Joe once they're built. These personas are *use cases*, and not meant to represent completely distinct individuals.

1.2. An example

To give the idea, here is a simple example. Angela has written a couple of Haskell modules that implement sets and bags; she wants to distribute them to Bob as a package called, say, `angela-coll`. Let's say that the modules are `Data.Set`, `Data.Bag`, `Angela.Internals`. (The Cabal says nothing about how Angela decides where in the name space to put her modules.) Angela only wants to expose the first two to Bob; the `Angela.Internals` module is (as its name suggests) internal to the package.

Angela decides to use the simple build infrastructure that the Cabal provides. She is working in a directory `~/coll`. In there she puts the modules, in sub-directories driven by their module name: `~/coll/Data/Set.hs`, `~/coll/Data/Bag.hs`, and `~/coll/Angela/Internals.hs`. Next, she writes a *package description*, which she puts in `~/coll/Setup.description`:

```
Name: angela-coll
Version: 0.1.1.1.1-foo-bar-bang
License: LGPL
Copyright: Copyright (c) 2004, Angela Author
Exposed-Modules: A, B, B.C
```

She also creates a small Haskell file `~/coll/Setup.lhs` as follows:

```
  #! /usr/bin/env runhugs

  > module Main where
  > import Distribution.Simple( defaultMain )
  > main = defaultMain
```

This library implements the Cabal simple build infrastructure.

The first line arranges that when Angela, (or Joe, or Sam, etc.) executes `Setup.lhs` as a shell script, the shell will invoke **runhugs**, which will in turn run `mainn` imported from the library `Distribution.Simple`.

It is not necessary that the script be run this way, it is just a convinient way to run it. Sam or Joe may choose to compile the setup script into an executable with NHC or GHC and then run it directly (it is a literate Haskell script so that it can be compiled without the first line causing a syntax error). Another option is for that first line to read

```
!# /usr/bin/env runhaskell
```

where **runhaskell** is a symlink to **runhugs**, **runghc**, or **runnhc**.

Now she is ready to go. She types:

```
  ./Setup.lhs configure --ghc
  ./Setup.lhs build
  ./Setup.lhs sdist
```

The first line readies the system to build the tool using GHC; for example, it checks that GHC exists on the system. The second line checks that the tool does indeed build flawlessly. (At this point she can write and execute tests, as we discuss later.) The third line wraps up the package as a source distribution, making the file `~/coll/angela-coll-1.tar.gz`.

Angela emails the tar file to Bob, who untars it into `tmp/coll`. He **cd**'s to that directory and types

```
./Setup.lhs configure --ghc
./Setup.lhs build
./Setup.lhs install
```

He's all done. Now in his Haskell programs, Bob can simply `import` the new modules `Data.Set` and `Data.Bag`. He does not need to give extra flags to GHC to tell it to look for Angela's modules; they are there automatically. If Angela used the same module names as someone else, Bob may need finer control: see Section 3.

If Angela wrote her modules in a suitably portable variant of Haskell, Bob could also have said `--hugs` or `--nhc` in his `configure` line, and the package would have been built and installed for those compilers instead.

2. The Haskell Package System: overview

This section summarises the vocabulary and main features of the Haskell Package System.

2.1. Packages

A *package* is the unit of distribution for the Cabal. Its purpose in life, when installed, is to make available some Haskell modules for import by some other Haskell program. However, a package may consist of much more than a bunch of Haskell modules: it may also have C source code and header files, documentation, test cases, auxiliary tools and whatnot.

Each package has:

- A globally-unique *package name*, containing no spaces. Chaos will result if two distinct packages with the same name are installed on the same system. How unique package names are handed out is not part of this specification, but there will presumably be some global web site where package authors can go to register a package name.
- A *version*, consisting of a sequence of one or more integers.
- *A list of explicit dependencies* on other packages. These are typically not exact; e.g. "I need `hunit` version greater than 2.4".

- *A list of exposed modules*. Not all of the modules that comprise a package implementation are necessarily exposed to a package client. The ability to expose some, but not all, of the modules making up a package is rather like using an explicit export list on a Haskell module.

The first two components can be combined to form a single text string called the *package ID*, using a hyphen to separate the version from the name, and dots to separate the version components. For example, "hunit-2.3".

2.2. Packages and the Haskell language

A complete Haskell program will consist of one or more modules (including `Main`) compiled against one or more packages (of which the Prelude is one). These packages are not referred to explicitly in the Haskell source; instead, the packages simply populate the hierarchical space of module names.

Complete programs must obey the following invariant. *Consider all the Haskell modules that constitute a complete program: no two modules must have the same module name.*

This invariant is conservative. It preserves the existing semantics of Haskell, and is relatively easy to implement. In particular, the the full name of an entity (type, class, function), which is used to determine when two entities are the same, is simply a pair of the module name and the entity name.

The invariant is unsatisfactory, however, because it does not support abstraction at the package level. For example, a module with an internal (hidden, non-exposed) module called `Foo` cannot be used in the same program as another package with an unrelated internal module also called `Foo`. Nor can a program use two packages, P and Q, which depend on different versions of the same underlying package R. We considered more sophisticated schemes, in which (for example) the package name, or package ID, is implicitly made part of every module name. But (a) there is a big design space, and (b) it places new requirements on the implementations. Hence a conservative starting point.

2.3. Packages and compilers

We use the term “compiler” to mean GHC, Hugs, Nhc98, hbc, etc. (Even though Hugs isn’t really a compiler, the term is less clumsy than “Haskell implementation”.)

The Cabal requires that a conforming Haskell compiler is somewhat package aware. In summary, the requirements are these:

- Each compiler **hc** must provide an associated package-management program **hc-pkg**. A compiler user installs a package by placing the package’s supporting files somewhere, and then using **hc-pkg** to make the compiler aware of the new package. This step is called *registering the package with the compiler*.
- To register a package, **hc-pkg** takes as input an *installed package description (IPD)*, which describes the installed form of the package in detail. The format of an IPD is given in Section 3.4.

- Subsequent invocations of **hc** will include modules from the new package in the module name space (i.e. visible to `import` statements).
- The compiler should support `-package` and `-hide-package` flags for finer-grain control of package visibility.

A complete specification of these requirements is given in Section 3.

2.4. Package distributions

A Cabal package can be distributed in several forms:

- A *Cabal source distribution* is a tree of files (tar-ball, zip file etc) containing the tool's sources, which may need to be compiled before being installed. The same source tarball may well be installable for several Haskell implementations, OSs, and platforms.

  A source distribution may contain fewer files than appear in the developer's CVS repository; for example, design notes may be omitted. It may also contain some derived files, that do not appear in the the developer's repository; for example, ones made by a somewhat exotic pre-processor where it seems simpler to ship the derived file than to ensure that all consumers have the pre-processor.

- A *Cabal binary distribution* is a tree of files that contains a pre-compiled tool, ready for installation. The pre-compilation means that the distribution will be Haskell-compiler-specific, and certain "looser" dependencies (`hunit > 2.3`) will now be precisely fixed (`hunit == 2.4`).
- The package may be wrapped up as an *RPM*, *Debian* package, or *Windows installer* (this list is not exhaustive). In that case, the way it is installed is prescribed by the respective distribution mechanism; the only role of the Cabal is to make it easy to construct such distributions. All three are compiler-specific (indeed compiler-version-specific) binary distributions.

2.5. The Setup script

The key question is this: how should Angela Author present her Cabal package so that her consumers (Bob, Sam, Willie, etc) can conveniently use it?

Answer: she provides a tree of files, with two specific files in the root directory of the tree:

- `Setup.description` contains a short description of the package: specifically, the package name, version, and dependencies. It may also contain further information specific to the particular build system. The syntax of the package description file is given in Section 4.1.
- `Setup.lhs` is an executable Haskell program which conforms to a particular specification, given in detail in Section 4. In summary, though, `Setup.lhs` allows a consumer to configure, build, test, install, register, and unregister a package.

The Setup script is an *interface*. It is meant to give a standard look-and-feel to packages for the sake of Joe User, Bob Builder, Peter Packager, Sam Sysadmin, and Rowland RPM, as well as for layered software tools. This interface provides an abstraction layer on top of any implementation that Angela or Marcus prefers.

The Cabal allows a package author to write the setup script in any way she pleases, provided it conforms to the specification of Section 4. However, many Haskell packages consist of little more than a bunch of Haskell modules, and for these the Cabal provides *the simple build infrastructure*, a Haskell library that does all the work. The simple build infrastructure, which was used for the example in Section 1.2, is described in Section 5.

In principle, the `Setup` script could be written in any language; so why do we use Haskell?

- Haskell runs on all the systems of interest.
- Haskell's standard libraries should include a rich set of operating system operations needed for the task. These can abstract-away the differences between systems in a way that is not possible for Make-based tools.
- Haskell is a great language for many things, including tasks typically relegated to languages like Python. Building, installing, and managing packages is a perfect proving ground for these tasks, and can help us to discover weaknesses in Haskell or its libraries that prevent it from breaking into this "market". A positive side-effect of this project might be to make Haskell more suitable for "scripting" tasks.
- Likewise, each piece of the project (Building, Installing, and Packaging) can be leveraged elsewhere if we make them into libraries.
- Make is not particularly good for parsing, processing, and sharing meta-information about packages. The availability of this information to Haskell systems (including compilers, interpreters, and other tools) is useful. Unlike Make, Haskell can also reuse unrelated algorithms, parsers, and other libraries that have been developed in the past.
- *Dogfooding*, the act of using the tools you develop, is a healthy policy.

It is convenient for consumers to execute `Setup.lhs` directly, thus:

```
./Setup.lhs ...
```

This can be achieved by starting `Setup.lhs` with "`#! /usr/bin/env runhugs`" or "`#! /usr/bin/env runghc`" . Since it's a literate Haskell script (`.lhs` file), the Haskell compiler will ignore this line. However, nothing stops a consumer from running the script interactively, or compiling it and running the compiled binary. Any implementation of Haskell should suffice to run the script, provided the implementation has the Cabal libraries installed.

3. What the compilers must implement

The Cabal requires that the Haskell implementations be somewhat package-aware. This section

documents those requirements.

3.1. Building and registering a package

Installing a package ultimately involves these steps:

- *Compiling the source files*, by invoking the compiler. Even Hugs may require some processing (e.g running cpp or cpphs).
- *Copying the compiled files into some permanent place*. Typically the compiler places no pre-conditions on where "some place" is; instead one usually follows the conventions of the host operating system.
- *Registering the package*: telling the compiler about the existence of the package, and where its files are. To register the package one invokes a compiler-specific program **hc-pkg** (i.e. **ghc-pkg**, **hugs-pkg** etc), passing it an *installed package description (IPD)* describing the package. The format of an IPD is given in Section 3.4.

It must be possible to register many versions of the same package.

3.1.1. Global packages and user packages

A package can be registered either as a *global package* or as a *user package*. The former means that anyone invoking **hc** will see the new package. The latter means that only the user who installed the package will see it.

User packages *shadow* global packages, in the following sense:

- A Haskell `import` for module M will seek M in a user package first.
- The **hc-pkg** commands that take package IDs will look for a user package first.

Each user has one package database per compiler and version. That is, the user packages for GHC 6.2 are separate from those for GHC 6.2.1. If there are multiple installations of a particular compiler version on the system, then they will all use the same user packages, so the user should refrain from using user packages with different installations of the same compiler version, unless they can guarantee that the different installations are binary compatible, such as if they were installed from identical binary distributions.

For instance, lets say that Joe User mounts his home directory on his Linux machine and on his Solaris machine. He also uses ghc-6.2 on both machines. His user packages are installed in ~/ghc-6.2-packages (or something) and are compiled for Linux. Now he had better not use these packages with the Solaris compiler, because they are in a spot where the Solaris compiler will find them!

3.1.2. Exposed packages and hidden packages

An installed package can be *exposed* or *hidden*. An exposed package populates the module name space, while a hidden package does not. Hidden packages are nevertheless necessary. For example, the user might use package A-2.1 and B-1.0; but B-1.0 might depend on A-1.9. So the latter must be installed (else B-1.0 could not be installed), but should be hidden, so that user imports see A-2.1. (However, note that the whole-program invariant described in Section 2.2 implies that a program using B-1.0 could not also use A-2.1, because then both A-2.1 and A-1.9 would form part of the program, and they almost certainly use the same module names.)

The registration program **hc-pkg** provides operations to expose or hide an already-installed package. By default, installing a package installs it exposed, and hides any existing installed package of the same name (and presumably different version).

Hiding or exposing a package is an operation that can be performed, by **hc-pkg**, on any package. It is quite distinct from the question of which modules in a package are hidden or exposed (see Section 2.1), which is a property of the package itself. Only the exposed modules of an exposed package populate the module name space seen by a Haskell `import` statement.

3.1.3. Registration invariants

The registration program **hc-pkg** checks the following invariants:

- Before registering a package P, check all the packages that P depends on are already registered. If P is being registered as a global package, P's dependencies must also be global packages.
- Before registering an exposed user package P, check that the modules that are exposed by P do not have the same names (in the hierarchical module name space) as any other module in an exposed user package Q. Similarly for system packages. (However, one may register a system package which exposes a module with the same name as a user package, and vice-versa.)
- Before un-registering a package P, check that no package that depends on P is registered. The exception is that when un-registering a global package, **hc-pkg** cannot check that no user has a user package depending on P.

3.2. The `-package` compiler flag

By default, the module namespace is populated only by the exposed modules of exposed packages. This can be overridden using the `-package` flag, which temporarily exposes a particular package, hiding any other packages of the same name.

Later options override earlier ones, so that for example `-package hunit-1.2 -package hunit-1.3` will result in `hunit-1.3` being exposed.

Additionally, compilers should provide a `-hide-package` flag, which does the opposite of `-package`: it temporarily hides a package for this run of the compiler.

When all the `-package` and `-hide-package` flags on the compiler's command line have been processed, the resulting module namespace must not contain any overlapping modules; the compiler should check this and report any errors.

3.3. The interface to hc-pkg

Registering a package with a compiler records the package information in some implementation-specific way; how it does so is not constrained by the Cabal. Much of an IPD is independent of the compiler, but it may also include compiler-specific fields.

Each Haskell implementation **hc** must provide an associated program **hc-pkg** which allows a user to make a new package known to the compiler, and to ask what packages it knows. Here is a summary of its interface

> **Note:** Some of these commands (unregister, hide, and describe) make sense for package IDs which offer a range, such as "hc-pkg unregister hunit<2.3".

Table 1. hc-pkg interface

hc-pkg register {*filename* | -} [--user | --global]	Register the package using the specified installed package description. The syntax for the latter is given in Section 3.4.
hc-pkg unregister [*pkg-id*]	Unregister the specified package.
hc-pkg expose [*pkg-id*]	Expose the specified package.
hc-pkg hide [*pkg-id*]	Hide the specified package.
hc-pkg list	List all registered packages, both global and user, hidden and exposed.
hc-pkg describe {*pkg-id* }	Give the registered description for the specified package. The description is returned in precisely the syntax required by **hc-pkg register**.
hc-pkg field {*pkg-id* } {*field* }	Extract the specified field of the package description for the specified package.

A *pkg* argument can be a package ID, such as "`hunit-2.3`", or just a package name, such as "`hunit`". To determine which package is meant, **hc-pkg** searches first the registered user packages and then the global packages. If no such package exists, the command fails; that is, it does nothing, returning a

non-zero error code. If only a name is specified, **hc-pkg** fails unless the name identifies a unique package among the user packages, or among the global packages. As usual, the user packages win.

> **Note:** Can we give the `--user` flag to **hide**, **expose**, **describe**? Can we register a package that is already registered? What if it's registered as a global package and we register it as a user package?

3.4. Syntax of installed package description

> **Note:** ...include the list of "externally visible modules".

4. The setup script

The sole requirement of an Cabal package is that it should contain, in the root of its file structure, (a) a package description file `Setup.description`, and (b) a setup script, `Setup.lhs`. This section specifies the syntax of the package description, and the command-line interface for the setup script.

4.1. The package description

Here is a sample package description file:

```
-- Required
Name: Cabal
Version: 0.1.1.1.1-rain
License: LGPL
Copyright: Free Text String
-- Optional - may be in source?
Stability: Free Text String
Build-Depends: haskell-src, HUnit>=1.0.0-rain
Modules: Distribution.Package, Distribution.Version,
         Distribution.Simple.GHCPackageConfig
...
```

The Name, Version, License, and Copyright are compulsory.

The rest of the lines are optional. Any fields the system doesn't understand will be ignored (to allow addition of fields at a later date, or to allow Setup script authors to provide their own fields). Each field must begin with "Field-Name:".

Lines beginning with "--" will be considered comments and ignored.

Each field must begin with "Field-Name:". In order to allow for long fields, any line that begins with whitespace will be considered part of the previous field.

As given, there should be no reason to have completely empty lines (except perhaps for the last line in the file). In the future, if there is need to add more records, they will be separated by at least one completely empty line.

For the Cabal-provided simple build infrastructure, the package description fields and syntax are given in Section 5.2.

4.2. The setup script specification

Here is the command-line interface the setup script must satisfy.

Table 2. setup.lhs interface

./Setup.lhs configure [flags]	Prepare to build the package. Typically, this step checks that the target platform is capable of building the package, and discovers platform-specific features that are needed during the build.
./Setup.lhs build	Make this package ready for installation. For a true compiler, this step involves compiling the Haskell source code. Even for an interpreter, however, it may involve running a pre-processor.
./Setup.lhs install [install-prefix]	Copy the files into the install locations, and register the package with the compiler.
./Setup.lhs register **./Setup.lhs unregister**	Register (or un-register) this package with the compiler. (NB: registration is also done automatically by `install`.)
./Setup.lhs clean	Clean out the files created during the configure, build, or register steps.
./Setup.lhs test	Run the package's test suite.

For wrapped make-based systems (for instance), a command-line parser that understands the standard `Setup.lhs` command-line syntax will be provided as a library.

4.2.1. `configure`

The command `./Setup.lhs configure` prepares to build the package. For sophisticated packages, the configure step may perform elaborate checks, to gather information about the target system. It may write a file to record its results, but the name and format of this file are not part of the specification.

All flags are optional. The flags are these:

- `--with-compiler=`*`path`*, `--ghc`, `--nhc`, `--hugs`: specifies which compiler to use. At most one of the value of these flags may be specified. The configure step checks that the compiler is available, in a sufficiently up-to-date form for the package, and that the package expects to work with that compiler. If the compiler name is not specified, `Setup.lhs` will choose one; some packages will come with one compiler baked in.
- `--prefix=`*`path`*: specifies where the installed files for the package should be installed (ie the location of the files themselves). Typically on Unix this will be `/usr/local` and on Windows it will be `Program Files`. The setup script will use a sensible default (often platform-specific) if the flag is not specified.
- Unrecognized flags are errors in the default build system, but may be meaningful to wrapped make-based systems (for instance). Therefore, the provided command-line parser will pass unrecognized command-line flags on to the wrapped system.

It is OK for these flags to be "baked into" the compiled tool. In particular, the build system may bake the installation path into the compiled files. There is no provision for changing these baked-into values after configuration.

4.2.2. `build`

The command `./Setup.lhs build` makes this package ready for installation. It takes no flags.

4.2.3. `install`

The command `./Setup.lhs install` copies files from the built package to the right location for installed files, specified in the configure step. Then it registers the new package with the compiler, using the **hc-pkg** command.

- `--install-prefix=`*`path`*: specifies where the installed files for the package should be installed (ie the location of the files themselves). It has three effects. First, it over-rides the `--prefix` flag specified in the `configure` step, providing an alternative location. Second, it does not call **hc-pkg** to register the package. Instead, third, it creates an installed package description file, `installed-pkg-descr`, which can later be fed to **hc-pkg**.
- `--user`: if present, this flag is passed to **hc-pkg** so that the package is registed for the current user only. This flag has no effect if `--install-prefix` is used, because in that case **hc-pkg** is not called.

- `--global`: if present, this flag is passed to **hc-pkg** so that the package is registed globally (this is the default if neither `--user` or `--global` are given). This flag has no effect if `--install-prefix` is used, because in that case **hc-pkg** is not called.

The reason for the `--install-prefix` flag is that Roland RPM wants to create an exact installation tree, all ready to bundle up for the target machine, *but in a temporary location*. He cannot use this location for `--prefix` in the `configure` step, because that might bake the wrong path into some compiled files. Nor does he want to register this temporary tree with the compiler on his machine. Instead, he bundles up the temporary installation tree, plus the `installed-pkg-descr`, and ships them all to the target machine. When they are installed there, the post-installation script runs **hc-pkg** on the `installed-pkg-descr` file.

Note that there is no **uninstall** command in the setup script. While it would be easy enough to implement in the simple build infrastructure, we don't want to require make-based build systems to implement `make uninstall`, which is fairly non-standard and difficult to get right. In the majority of cases, we expect libraries to be installed via a package manager (eg. RPM, Debian APT), which already provide uninstallation services.

4.2.4. `register` and `unregister`

The command `./Setup.lhs register` registers the now-installed package with the compiler. Similarly, `./Setup.lhs unregister` un-registers the package.

- `--global`: registers/un-registers a package as global. This is the default.
- `--user`: registers/un-registers a package for the current user only.

4.3. Examples

4.3.1. Bob the Builder and Sam Sysadmin

Bob the Builder can install a Cabal source distribution thus. He downloads the source distribution and unpacks it into a temporary directory, `cd`'s to that directory, and says

```
./Setup.lhs configure --ghc
./Setup.lhs build
./Setup.lhs install --user
```

Similarly, Sam Sysadmin does exactly the same, except that he says

```
./Setup.lhs install --global
```

in the final step, so that the package is installed where all users will see it.

For a binary distribution, both Bob and Sam would omit the first two steps, and just do the install step.

4.3.2. System packagers (Debian, RPM etc)

System packagers, such as Peter Packager or Donald Debian, will run the configure and build steps just like Bob and Sam. A that point, Donald will say

```
./Setup.lhs install --install-prefix=/tmp/donald
```

to construct a ready-to-zip tree of all the installed files, plus a file `installed-pkg-descr` that describes the installed package. He arranges to deliver both these components to the target machine, and then feed `installed-pkg-descr` to **hc-pkg** on the target machine.

The file `installed-pkg-descr` also contains information he needs for building his Debian distribution, namely the package name, version, and (exact) dependencies.

We expect there to be additional tools to help System Packagers to prepare the materials necessary to build their packages from a source distribution. For example, an RPM tool could take a Haskell package source distribution and build an initial `.spec` file with as many of the fields as possible filled in automatically. In most cases some intervention by the System Packager will be necessary; for example platform-specific dependencies may need to be specified.

After Peter has constructed the package, Isabella can install it in a manner she is comfortable with.

5. The Cabal simple build infrastructure

A package author must fulfil the specification of Section 4. In many cases, a Haskell package will consist of nothing more than a bunch of Haskell modules, with perhaps the odd C file. In that case, the Cabal provides a *simple build infrastructure* that fulfils the specification of Section 4, and provides some modest further facilities besides.

This simple build infrastructure is meant to automate the common case. (Think **hmake**.) The emphasis is on “simple”: if you want something more elaborate, you can (a) modify the simple build infrastructure (which is written in Haskell) (b) use makefiles, or (c) implement something else entirely.

5.1. Overview

The simple build infrastructure works as follows. First, Angela puts the following Haskell file

`Setup.lhs` in the root of her tree:

```
#! /usr/bin/env runghc

> import Distribution.Simple
```

Second, she writes a package description `Setup.description` in the syntax of Section 5.2, which describes the package and gives extra information to the simple build infrastructure.

Now Angela can build her package by saying

```
./Setup.lhs configure
./Setup.lhs build
```

She can even install it on her own machine by saying

```
./Setup.lhs install
```

She can build a Cabal source distribution:

```
./Setup.lhs source-dist
```

The full details are given in Section 5.3.

It is no coincidence that the interface is very similar to that for the setup script for an Cabal package distribution (Section 4). In fact, `Distribution.Simple.defaultMain` conforms to the specification of Section 4.2, and when it builds a distribution, it includes `./Setup.lhs` in the tarball, ready to be run by Bob the Builder. However, `Distribution.Simple.defaultMain` of course implements a richer interface than that required by Section 4.2, because it's intended to support Angela as well as Bob. The full specification is in Section 5.3.

5.2. Package description in the simple build infrastructure

When using the simple build infrastructure, the package description file `Setup.description` contains not only the name of the package, its version and dependencies, but also a collection of information to explain to the simple build infrastructure how to build the package. This section gives the specific fields, and the syntax of those fields. For the general syntax of the file, please see Section 4.1.

Here is a sample package description file with all the fields understood by the simple build infrastructure:

```
-- Required
Name: Cabal
Version: 0.1.1.1.1-rain
License: LGPL
Copyright: Free Text String
-- Optional - may be in source?
Stability: Free Text String
Build-Depends: haskell-src, HUnit>=1.0.0-rain
```

```
Modules: Distribution.Package, Distribution.Version,
         Distribution.Simple.GHCPackageConfig
C-Sources: not/even/rain.c, such/small/hands
HS-Source-Dir: src
Exposed-Modules: Distribution.Void, Foo.Bar
Extensions: OverlappingInstances, TypeSynonymInstances
Extra-Libs: libfoo, bar, bang
Include-Dirs: your/slightest, look/will
Includes: /easily/unclose, /me, "funky, path\\name"
Options-ghc: -fTH -fglasgow-exts
Options-hugs: +TH

-- Next is an executable
Executable: somescript
Main-is: SomeFile.hs
Modules: Foo1, Util, Main
HS-Source-Dir: scripts
Extensions: OverlappingInstances
```

The Name, Version, License, and Copyright are compulsory.

All other fields, such as dependency-related fields, will be considered empty if they are absent. Any fields that the system does not understand will be ignored.

Note that in the future, though the Modules field will be available, it will not be necessary to provide it for building executables and libraries. Instead, the user will provide only the "Main-Is" field (for executables) and the "Exposed-Modules" field (for libraries). The system will chase down dependencies from those modules and include them in the library or source distributions.

The description file fields:

Table 3. Description File Fields

Field Name	Description	Example	Notes
name	[a-zA-Z][-a-zA-Z0-9]*	haskell-cabal12345	
version	[0-9.]+(-?)[-a-zA-Z]*: branch numbers, separated by dots, and optional tags separated by dashes.	1.2.3.4.5-foo-bar-bang	
copyright	--FREE TEXT--	(c) 2004 Isaac Jones	
license	GPL | LGPL | BSD3 | BSD4 | PublicDomain | AllRightsReserved	BSD3	If your license isn’t on this list, use the *license-file* field.

license-file	--PATH--	doc/myLicense.txt	Specify the license you use as a relative path from the root of the source tree.
maintainer	--FREE TEXT--	T.S. Elliot <elliot@email.com>	
stability	--FREE TEXT--	Don't hook this up to your coffee machine	
executable	--FREE TEXT--	cpphs	For this Executable stanza, what is the name of the produced executable.
main-is	--PATH--	/foo/bar/bang/Baz.hs	The filename to look for the main module for this Executable stanza.
extra-libs	comma list of --FREE TEXT-- and spaces	libfoo, libbar , libbang	for non-haskell libraries that this package needs to link to
build-depends	package name (== | < | > | <= | >=) version	foo > 1.2, bar < 3.3.5, bang	If the version isn't listed, it's assumed any version is OK.
c-sources	--PATH--	/foo/bar/bang	C source files to build using the FFI.
include-dirs	--PATH--	"/foo/bar/ ,bang"	Not Yet Used
includes	--PATH--	/foo/bar/bang	Not Yet Used
hs-source-dir	--PATH--	src	A relative path from the root of your source tree. Look here for the Haskell modules.
modules	--MODULE LIST--	Foo.Bar, Bang.Baz, Boo	May not be necessary in the future, since we'll chase dependencies from exposed modules and main module.
exposed-modules	--MODULE LIST--	Foo.Bar, Bang.Baz, Boo	For a library package, which modules should be available for import by the end user?

extensions	OverlappingInstances | RecursiveDo | ParallelListComp | MultiParamTypeClasses | NoMonomorphismRestriction | FunctionalDependencies | RankNTypes | PolymorphicComponents | ExistentialQuantification | ScopedTypeVariables | ImplicitParams | FlexibleContexts | FlexibleInstances | EmptyDataDecls | TypeSynonymInstances | TemplateHaskell | ForeignFunctionInterface | AllowOverlappingInstances | AllowUndecidableInstances | AllowIncoherentInstances | InlinePhase | ContextStack | Arrows | Generics | NoImplicitPrelude | NamedFieldPuns | ExtensibleRecords | RestrictedTypeSynonyms | HereDocuments | UnsafeOverlappingInstances	ForeignFunctionInterface, Arrows	Not all extensions are understood by all Haskell Implementations
options-ghc	--OPTIONS--	-fth -cpp	For command-line options not covered under *extensions*, add them here, separated by whitespace.
options-nhc	--OPTIONS--	-P -t	For command-line options not covered under *extensions*, add them here, separated by whitespace.
options-hugs	--OPTIONS--	-98 +g	For command-line options not covered under *extensions*, add them here, separated by whitespace.

Further details on some fields:

- --*PATH*-- Paths are written in the Unix style, with directories separated by slashes, optionally ending in a filename. There are two kinds of paths supported: "Simple" and "Complex". "Simple" paths are alpha-numeric values separated by slashes (foo/bar/bang). More "Complex" paths such as those with spaces or non-alphanumeric characters must be put in quotes ("foo, /bar \//bang"). You should assume that the paths are case sensitive, though in practice this varies depending on the end user's file system.
- --*MODULE LIST*-- A module list is a standard Haskell module name, without any file suffixes (.lhs or .hs). Each module should be separated by a comma.
- --*FREE TEXT*-- You may put anything in these fields. For multi-line fields, start the subsequent lines with whitespace.
- --*OPTIONS*--The exact syntax of the options field is dependent on the compiler you're using. Refer to your compiler's man page for details. Multiple options should be separated by whitespace.

On hidden modules: Hidden modules form part of the implementation of the package, but not its interface: a client of the package cannot import an internal module. The system still must derive their existence, or they must be listed explicity for two reasons: (a) to allow the global program invariant to be checked (see Section 2.2) and (b) to enable a build system or programming environment to find the source files.

5.3. `Distribution.Simple`

This section gives the command line interface supported by `Distribution.Simple.defaultMain`. It supports all the commands described in Section 4.2, (except for "test" - FIX) and in addition the following:

Table 4. Extra commands supported by the simple build infrastructure setup script

./Setup.lhs sdist	Create a source tarball
...	...

Distribution.Simple.defaultMain provides interactive command-line help. For each command, a help string is available by typing "./Setup.lhs COMMAND --help".

5.4. The Makefile route

The Haskell libraries that support the simple build infrastructure can, of course, also be re-used to make

setup scripts that work quite differently. At one extreme is a setup script that immediately shells out into `make`, which does all the work.

To support this, Cabal provides a trivial setup library `Distribution.Make`, which simply parses the command line arguments and shells out into `make`. Marcus uses the following `Setup.lhs`

```
#! /usr/bin/env runhugs

> module Main where
> import Distribution.Make (defaultMain)
> main = defaultMain
```

All the package description information is assumed to be known to the makefile system, and so does not appear in the setup script. Thus,

```
./Setup.lhs configure --ghc
```

invokes

```
./configure --with-hc=ghc
```

Similarly `./Setup.lhs build` invokes `make all` And so on.

Marcus simply arranges that when his makefiles build a distribution, they include this simple setup script in the root of the distribution, where the Bob the Builder expects to find it.

A. Layered Tools

One great advantage to having a standard configure/build/install interface for all Haskell packages is that end users don’t need to learn a new method for each package they install.

Likewise, with such an interface, we can build layered tools in a typical Unix fashion. For instance, we might want a tool that downloads and installs Haskell packages. It can be implemented as a simple shell script, haskell-install (where "$@" represents the command-line argument):

```
#!/bin/sh

echo "wget http://packages.haskell.org/$@"
echo "tar -zxvf $@.tgz"
echo "cd $@"
echo "./Setup.lhs configure"
echo "./Setup.lhs build"
echo "./Setup.lhs install"
```

Now the end-user (Isabella) only needs to say "haskell-install myPackage", rather than performing all of the Setup steps by hand.

B. Related Systems

I will try to outline interesting points in a variety of systems that we can learn from. These systems may be intended for building or installing packages, or repositories for packages. I am not deeply familiar with all of the tools here, and would be interested in hearing more relevant points from someone with more knowledge. Another weakness of mine is that I don't know much about Microsoft Windows, so some good examples for Windows systems would be helpful.

B.1. Debian

The Debian GNU/Linux system (http://www.debian.org) is a good example of a *binary* distribution (meaning that packages are distributed in binary, as opposed to source code form), and its packaging system (dpkg) is somewhat similar to the more famous RPM. Debian has several other tools to help the user to install packages, most notably, **apt**. The Debian toolset is interesting for several reasons:

- It handles dependencies extremely well. A single command can download and install a package, as well as downloading and installing all of its dependencies.
- It handles updates extremely well. One command (**apt-get update**) checks for new versions of packages and updates a local database. Another command (**apt-get dist-upgrade**) downloads and installs all new versions of installed packages and any new dependencies.
- There are standard commands for downloading and building packages from source. If I'm interested in hacking on a package, I can run **apt-get source packagename** which will download and unpack the source code for the package. The source can then be built with the standard command **debuild**.
- The Debian Project maintains a central repository for packages, and the packaging tools offer support for using unofficial repositories as well. The central repositories include a set of servers, the *autobuilders*, which compile uploaded source packages for a variety of hardware architectures (see below) and make them available as binary packages. As a packager, I merely upload the source code to my package, and the autobuilders do the rest.
- Currently the hardware architectures supported by Debian are Intel x86, Motorola 68k, Sun SPARC, Alpha, PowerPC, ARM, MIPS, HP PA-RISC, IA-64, S/390. Debian also runs on non-Linux systems, including GNU/Hurd, GNU/NetBSD, and GNU/FreeBSD. The package management tools also run on MacOS X under the name of the Fink project.

B.2. Python Distutils

Python's Distutils system (http://www.python.org/sigs/distutils-sig/) is in many ways similar to what we propose here. It is a system for building and installing Python modules, written purely in Python. The user interface is a Python script, (`setup.py` by convention) and a setup configuration file (`setup.cfg` by convention). To quote from Distributing Python Modules (http://www.python.org/doc/current/dist/dist.html), "The setup configuration file is a useful middle-ground between the setup script--which, ideally, would be opaque to installers -- and the command-line to the setup script, which is outside of your control and entirely up to the installer. "

Its noteworthy that Python has a big advantage over many programming languages when implementing a system like Distutils: It is designed to be well suited to so-called scripting tasks, which are common to the installation task, and Python has done these tasks in a portable way for a long time. I believe that Haskell should evolve portable ways to perform common scripting tasks.

B.3. CPAN and Boost

Quoting from CPAN's web site (http://www.cpan.org) "CPAN is the Comprehensive Perl Archive Network, a large collection of Perl software and documentation." That really says it all. It is a central location where Perl developers can contribute the software they write.

CPAN has a means of standardizing installation, `Makefile.pl` (which is a Perl script which creates a Makefile with targets like "install", "test", "config", "clean", etc.). Makefile.pl typically uses the MakeMover module (http://www.perldoc.com/perl5.6/lib/ExtUtils/MakeMaker.html). It also has a means of registering a namespace for the module that a developer is contributing.

From the Boost web site (http://www.boost.org) "[Boost] provides free peer-reviewed portable C++ source libraries. The emphasis is on libraries which work well with the C++ Standard Library. One goal is to establish "existing practice" and provide reference implementations so that the Boost libraries are suitable for eventual standardization. Some of the libraries have already been proposed for inclusion in the C++ Standards Committee's upcoming C++ Standard Library Technical Report."

From what I can tell, unlike CPAN, Boost is a bit more focused on standards and review. That is, it is perhaps more Cathedral than Bazaar [1]. Boost does not currently have a standard means of installation.

B.4. FreeBSD's Ports System

The FreeBSD Ports Collection (http://www.freebsd.org/ports/) is a collection of software with a standard means of compilation and installation. FreeBSD is a source distribution (whereas Debian is a Binary Distribution). Packages come in source-code form with a Makefile suitable for installing the program on a FreeBSD system. The ports collection is very large (around 9000 packages).

Some things may be simpler with a source distribution than with a binary distribution. For instance, since the code is expected to be already on the machine and buildable, when a new compiler is installed one merely needs to rebuild the dependant libraries. In contrast, with a binary distribution like Debian one must wait for a new binary package to be made available. However, as I understand it, FreeBSD has no means of recompiling dependant packages automatically when a new compiler is installed.

B.5. The XEmacs Packaging System

As most folks know, XEmacs is not only a text editor, but also a Lisp environment. Its functionality can be extended with lisp programs, and many such programs are available from XEmacs' Packaging System (http://www.xemacs.org/Documentation/21.5/html/lispref_4.html). Simply put, the packaging system offers a menu-driven interface within XEmacs where the user can browse available packages, select packages she is interested in, and ask XEmacs to download and install them. This system is interesting because it is cross-platform (Unix, Linux, Windows, etc.) and is designed to work only with elisp.

B.6. Make-Based Systems

The "fptools" build system has been used for many years in the cross-platform GHC compiler. It is a make-based system which is capable of a wide variety of installation tasks, compilation tasks, and system configuration tasks. Currently, it is not entirely generic across Haskell Implementations, and does not yet deal with some of the package registration issues mentioned above.

At Yale, another system is being developed. It is also a make-based system and works reasonably well on various platforms (Unix, Linux, Windows) and Haskell Implementations. It also does not yet deal with all of the package registration issues mentioned above.

Both tools can benefit from a standard packaging system.

Because make has been used for many years, it is expected that these systems will be able to do more than the initial release of the `Distribution` module. The Setup script will be designed with this in mind, and should be able to wrap these tools in order to provide a common interface for users and for layered tools.

B.7. hmake

From the hmake home page (http://www.cs.york.ac.uk/fp/hmake/), "hmake is an intelligent compilation management tool for Haskell programs. It automatically extracts dependencies between source modules, and issues the appropriate compiler commands to rebuild only those that have changed, given just the name of the program or module that you want to build. Yes, you need never write a Makefile again!" hmake also does a good job of handling the variety of compilers that might be installed on a user's

system. It maintains a list of compilers and can switch between them according to a flag. It also has a default compiler.

hmake is particularly interesting to us because it is written in Haskell and handles the task of compiling Haskell tools quite well. One shortcoming is that it is not extensible on a per-project basis: it is difficult to add support for new preprocessors without editing the hmake code itself. It does, however, perform a lot of the tasks that `Distribution.Build` will ultimately have to perform, and we hope to reuse some of the code.

Another interesting feature of hmake is the Haskell Interactive tool (hi). hi "is, an interpreter-like environment that you can wrap over any common Haskell compiler to achieve an interactive development style." This is interesting because it would be nice to have a generic `/usr/bin/haskell` which would use the default compiler to interpret Haskell scripts.

Notes

1. See Eric Raymond's essay The Cathedral and the Bazaar (http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/).