The Haskell Cabal

A Common Architecture for Building Applications and Libraries

Isaac Jones

Galois Connections

Abstract

Many programming languages and operating systems have a means of bundling together related source files, libraries, documentation, or executables for distribution. Such packaging systems contribute to a developer's ability to write complex software, since they can build upon reusable components which can be easily installed on an end-user's system. In addition, ease of installation can greatly increase initial adoption of a piece of software in user communities.

The Haskell Cabal is an architecture for building and installing any tool developed in the Haskell language. It specifies a command-line interface that makes it easy for end users to build and install libraries and applications, and supports tool authors by providing a library which implements this interface.

Cabal draws inspiration from a wide variety of tools and may in turn inspire tools for distribution of software in other functional programming languages. It is an evolving piece of open source software, and as such, we welcome discussion and contributions from any user community with an interest in its direction.

Cabal was officially released with the latest version of three Haskell compilers, *GHC* 6.4, *Hugs98* March 2005, and *nhc98* 1.18. [1] Its interface has already been used to implement a number of layered tools, and many newly released software tools use Cabal's API as their build system.

1 INTRODUCTION

The Haskell Cabal is an architecture for building and installing any tool developed in the Haskell language. It thereby facilitates distribution of any such tool. Cabal accomplishes three main goals:

1. Capture meta-data related to the package, including information about how to build it and other information such as a human-language description of the package and its license. See section 2.1 for more information about package meta-data.

2. Specify a standard command-line interface (CLI) for configuring, building, and installing packages. Since all packages conform to the same interface, end users can interact with packages in a familiar way. Also, tools can be layered on top of Cabal using the CLI. A number of such tools have already been released. See section 2.2 for more details about the CLI.

[1] Newer versions may be available from the Cabal web site, `http://www.haskell.org/cabal`.

3. Provide a library for configuring, building, and installing packages so that authors can easily generate packages which conform to the CLI. This infrastructure is flexible and extensible in a variety of ways. See section 2.3 for more information on the build infrastructure.

Together, the meta-data, the API, and the standard build and installation interfaces allow other tools to interact with Cabal packages in a predictable way.

The basic use-case for Cabal is this: The author of a software tool (a "package") creates a file which contains meta-data about their software. They then distribute the package bundled together with the meta-data just like any piece of software. The end-user downloads the package and uses Cabal to build and install it from source, based on the meta-data provided by the author.

Since Cabal has different interfaces depending on the user's point of view, in this paper, we will refer to the user who creates a Cabal package as the "packager" and the user who uses this package to install a piece of software on their computer as the "end-user". We will refer to Haskell implementations as "compilers" even though they may be compilers such as *GHC*, or interpreters such as *Hugs98*.

This paper outlines the problems that Cabal helps to solve and highlights design decisions that have made Cabal flexible and easy to integrate with a large number of tools. It also argues that Haskell is suitable as a "glue" language in areas where Perl, Python, or shell scripting languages are more typically deployed.

1.1 Background on Packages

Many programming languages and operating systems have a means of bundling together related source files, libraries, documentation, or executables for distribution. Such bundles are often called *packages*.

A *Cabal package* is a collection of Haskell modules distributed together, typically designed to perform a task or related set of tasks. For instance, the package *HUnit* [18] is a unit testing framework and the package *HaXR* [11] is an API which implements the remote procedure calling infrastructure xml-rpc [30].

The idea of a package exists in Haskell because it is often the case that developers and users of Haskell modules prefer to distribute and reason about related collections of modules together rather than as individual modules.

A Cabal package is a collection of Haskell modules described by a declarative, compiler-independent *package description file* (typically `<PackageName>.cabal`) and a *setup script* (typically `Setup.lhs`). A package may include a single library, any number of executable programs, or both. Packages are typically distributed to end-users by compressing them and rolling them into a single file with tools like tar and gzip.

There are other sorts of packages that have interesting relationships with Cabal packages. Many operating systems include a package management tool such as RPM or dpkg. We will distinguish such packages from Cabal packages by referring to them as "OS packages" or "RPM packages", for instance. Haskell compilers

may also have their own concept of a package which is closely related to a Cabal package.

1.2 Challenges of packaging in Haskell

When the author of a library or tool sets out to bundle together a set of Haskell modules, there are a number of challenges to face. One major challenge is that there are several different Haskell compilers, and end users may prefer one over another. Even if their tool is written in portable Haskell, without Cabal it can be difficult for the author to make it easy to build and install for end-users of each system.

For instance, *Hugs98* is an interpreter with no strong concept of a package, and *GHC* is a compiler which does have such a concept. Furthermore, both of these Haskell compilers are portable across a number of operating system platforms (including various Unix implementations, Linux, and Windows). Each operating system may have different tools available on them.

An author of a relatively simple Haskell library may find that a build system which works for both *Hugs98* and *GHC* on Windows and Linux will be more complex than the piece of software they would like to distribute.

In the Unix world, when one sets out to build a non-trivial collection of code, the first tool which comes to mind (after the compiler, perhaps) is *make* [20]. However, *make* presents a number of problems when trying to construct a reusable build architecture. See section 6.1 for more information. Besides providing a means of packaging software, Cabal sets out to obviate *make* for most Haskell tools.

2 IMPLEMENTATION DETAILS

Cabal provides a *simple build infrastructure* (see section 2.3) which can be used to configure, compile, install, and package many Haskell tools. This infrastructure is implemented entirely in Haskell.

Cabal specifies two interfaces for packagers: A command-line interface for building layered tools, and an API which provides programmatic access to the simple build infrastructure.

2.1 Package Description File

Any tool written in the Haskell language will have some unique information, such as its name, version number and a description of what it can be used for. Cabal requires that the packager capture this information in a simply formated *package description file*. Typically, this file will have the name `<PackageName>.cabal`.

The package description file holds information about how to build the package, as well as other meta-data which may be of interest to end-users or package databases. It is compiler-independent; the same description file should work for

```
Name            : pkgName
Version         : 1.0
Build-Depends   : OtherPackage >= 1.1
Copyright       : (c) 2005 Isaac Jones
License         : BSD3
Maintainer      : Isaac Jones
Synopsis        : Brief description.
Exposed-Modules: MyPackage
Other-Modules   : MyPackage.SubModule,
                  MyPackage.OtherModule
C-Sources       : cfile1.c, c_src/cfile2.c
Extensions      : ForeignFunctionInterface

Executable      : packageExe
Main-is         : Main.hs
Other-Modules   : MyPackage
Extensions      : OverlappingInstances
```

FIGURE 1. A package description file

each Haskell compiler that Cabal supports. It corresponds to the `PackageDescription` type in the Cabal API.

Figure 1 is an example package description file for a package named `pkgName`. `pkgName` has a library, which Cabal will generate and call `libHSpkgName-1.0.a` as well as an executable which will be called `packageExe`, as specified by the Executable field. Version numbers are used to specify precise dependencies and to allow multiple versions of packages to be installed.

2.2 Command-line interface

The highest-level interface, from the perspective of a tool layered on top of Cabal, is the command-line. A Cabal-compatible build system must conform to this interface; it is a standard "look and feel" for installing packages. This helps end-users to more easily install software by hand, and also allows one to build layered tools which operate on this interface.

For instance, one might imagine a very simple tool which fetches packages from the internet, then compiles and installs them on an end-user's machine such as the following pseudo-code:

```
setupCmd c = system ("./Setup.lhs " ++ c)
main = do
  [package] <- getArgs
  download "http://packages.haskell.org/"
           package
```

```
unpack package
changeDirectory package
setupCmd "configure --prefix=/usr/local"
setupCmd "build"
setupCmd "install"
```

Since each Cabal package is guaranteed to have `configure`, `build`, and `install` commands, this script will work for any package.

In fact, the interface is not quite that simple, but it is close. Each Cabal package must provide a module, `Setup.hs` or `Setup.lhs` which, when run, provides the following commands:

`configure`: Prepare to build package. Must be run before `build`.

`build`: Make package ready for installation. This must be run before `install`. This command will run any necessary preprocessors before proceeding. For *GHC*, this command will compile the package. For *Hugs98*, it will only prepare the source code for installation.

`install`: Copy compiled files into the correct locations on the end-user's machine and if there is a library, register the package with the compiler.

`copy`: Copy the files into the install locations, as in `install`, but do not register the package.

`register`: If there is a library in this package, register it with the compiler so that the compiler knows where to find the library.

`unregister`: Unregister this package with the compiler.

`clean`: Clean up after a build.

Each command may have flags to further refine its behavior. For instance, many of the commands provide a `--user` flag to indicate whether this package should use a system-wide package database, or a package database local to the end-user. This allows packages to be installed in an end-user's home directory.

It is suggested that the packager provide a `Setup.lhs` script with a Unix #! instruction at the top so that the Haskell script can be directly interpreted as above.

2.3 Simple Build Infrastructure

In order to make the job of creating a package as simple as possible, Cabal provides a system for configuring, building, and installing Haskell libraries and tools. For many tools, a packager merely has to provide a few fields in the package description file, and Cabal will be able to build the package on a variety of operating system platforms and Haskell compilers. This is typically much simpler than writing a `Makefile`, since the complexity of this task is captured in a reusable library.

During the *configure* step, Cabal will locate the compiler as well as any other package which this package requires in order to compile. For packages with more complex configuration needs *configure* may execute an autoconf [1] script (`./configure`) if available.

During the *build* step, Cabal will run any preprocessors that this package needs and possibly compile the Haskell source files. The build step may produce a set of

executables or a compiled library, `libHSpkgName-1.0.a`. It may not need to compile anything if this package has been configured to use the *Hugs98* interpreter. Cabal has knowledge of which flags to pass to which compilers during the build phase based on the metadata provided in the package description file.

During the *install* step, Cabal moves the library, executables, or source files into place, and may "register" the package with the compiler.

2.4 The Setup Script

The packager must provide a module `Setup.hs` or `Setup.lhs` which implements the command-line interface (see section 2.2). For most packages, Setup will be very simple:

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

> import Distribution.Simple(defaultMain)
> main = defaultMain
```

As mentioned in section 2.2, `defaultMain` implements the command-line interface necessary for compatibility. Cabal requires `Setup.lhs`, however, because the simple build infrastructure may not be able to build all packages; extra steps may be necessary, and only the packager can know in advance what these steps are.

In fact, `Setup.lhs` can perform whatever actions the packager wants it to perform as long as it conforms to the command-line interface. This means that arbitrarily complex packages, or packages which use a completely different build system, can still conform to the Cabal interface by creating a wrapper Setup module. Packages do not necessarily have to use the simple build infrastructure API.

2.5 API

The API provides programmatic access to the simple build infrastructure (see section 2.3), and consists of about twenty exposed modules in the `Distribution.*` hierarchy. Most Cabal users will only be interested in a handful of types and functions in the simple build infrastructure exported by the module `Simple`. The module is so named because it was originally designed to only operate on very simple modules, but the needs of the community quickly asserted themselves and it can now handle modules with a variety of extensions and preprocessors, including FFI and profiling. However, its interface remains very simple: `defaultMain :: IO ()`

`defaultMain` is often the only function that a packager will need. It implements the command-line interface described in section 2.2. It parses commands and options entered by the end-user and executes operations like `configure`, `build`, and `install`.

```
defaultMainWithHooks :: UserHooks -> IO ()

data UserHooks = UserHooks
  {
   hookedPreProcessors :: [ PPSuffixHandler ]
   {- ^Custom preprocessors in addition to
    and overriding built-in preprocessors -}

     -- |Hook to run before configure
   ,preConf  :: Args
             -> ConfigFlags
             -> IO HookedBuildInfo

    -- |Hook to run after configure
   ,postConf :: Args
             -> ConfigFlags
             -> LocalBuildInfo
             -> IO ExitCode  ... }
```

FIGURE 2. The UserHooks type

Such functions are also made available by the API so that developers can build tools that use the API rather than the command-line interface, and so that packagers can write more complex Setup modules.

`Distribution.Simple` is not quite that simple, however. It also has a flexible system for extensibility. The type `UserHooks` in Figure 2 is provided so that the packager can customize the behavior of the commands and preprocessors. For instance, if there is a preprocessor that Cabal doesn't know about, the packager can add a custom preprocessor hook (see section 2.6 for more on preprocessors). If there is some action that needs to be taken before or after a given command, for instance calling an external configure script before the internal configure step, the UserHooks are the place to do it.

Each package comes with a description file (see section 2.1). This description is implemented by the `PackageDescription` module. Some of the important types from that module are shown in Figure 3.

2.6 Pre-Processors

Preprocessors, tools which input various kinds of source files and output Haskell code, are common in Haskell packages. Cabal can automatically handle a variety of preprocessors, including Greencard [23], C2hs [12], Hsc2hs [6], Alex [13], Happy [14], and cpphs [27]. Packagers can extend their Setup modules to handle other preprocessors by providing *preprocessor hooks* (`PPSuffixHandler`.)

```
data PackageDescription =  PackageDescription {
        package        :: PackageIdentifier,
        license        :: License,
        copyright      :: String,
        maintainer     :: String,
        homepage       :: String,
        synopsis       :: String,
        buildDepends   :: [Dependency],
        -- components
        library        :: Maybe Library,
        executables    :: [Executable]
        ...
     }
data Library = Library {
        exposedModules    :: [String],
        libBuildInfo      :: BuildInfo
     }
data Executable = Executable {
        exeName    :: String,
        modulePath :: FilePath,
        buildInfo  :: BuildInfo
     }
```

FIGURE 3. Types from the PackageDescription module

3 LAYERED TOOLS

One purpose of specifying a command-line interface and API is so that developers can write tools which build on top of Cabal in order to extend its capabilities. The following sections describe some tools which use Cabal's API, command-line interface, or meta-data from package description files.

3.1 Hackage Database and cabal-get

HackageDB is the Haskell Package database. A packager can upload a tarred and compressed package with a package description file in it, and HackageDB will parse the description file using Cabal's API. HackageDB then integrates the package's meta-data into its database and provides a web interface for end-users to search for packages they may be interested in. HackageDB then saves the package locally so it can be downloaded later. HackageDB also provides an XML-RPC [30] interface so that a client-side tool can query the database for information about packages, including a URL to download them.

cabal-get is a tool for downloading and installing Cabal packages which are stored in the Hackage Database. It uses XML-RPC to interface with HackageDB, and also uses the Cabal API and the Cabal command-line interface to build and install packages and their dependencies.

A user simply runs: `cabal-get install packagename` and *cabal-get* will compute the packages which `packagename` depends on (based on the Build-Depends field in Cabal package description files), and download all of them. *cabal-get* then uses Cabal's standard "configure", "build", and "install" command-line targets to install each package. *cabal-get* is conceptually similar to the pseudo-code referenced in section 2.2, and is inspired by Debian's `apt-get`[22].

3.2 dh_haskell and cabal2rpm

Debian GNU/Linux [3] is a distribution of the GNU/Linux operating system. Its packaging system can automatically install and update packages and their dependencies. Similarly, Redhat Package Manager is a system for Redhat, Fedora, and other GNU/Linux-based operating systems. *dh_haskell* [15] and *cabal2rpm* [19] are tools that operates on a Cabal packages to produce a Debian or RPM packages respectively. They will work for any Cabal package which conforms to the Cabal command-line interface.

3.3 Visual Studio plugin for Haskell

Visual Studio is an integrated development environment for Windows. It supports a number of languages, and there is an ongoing project to build a Visual Studio plugin for Haskell [9]. Visual Studio's Haskell plugin uses Cabal's API as a build system. It can also read and write Cabal package description files in order to interpret them as Visual Studio projects. A pair of developers may therefore collaborate on a tool, one using Linux and one using Windows, and they will both feel that they are in a native environment.

4 DESIGN PRINCIPALS

A number of design choices made in Cabal may be interesting to functional programmers who seek to integrate a large number of tools, in various languages.

4.1 Command-Line Interface and Layered Tools

Although Cabal is designed to build most Haskell tools, the simple build infrastructure cannot take into account every possible usage scenario. Indeed, many of the libraries in the `fptools` build tree use `autoconf`, in addition to the built-in configure system, to discover information about the system they are building on.

Rather than providing only the simple build infrastructure, Cabal therefore also specifies a command-line interface. We chose this design principal to give packagers additional flexibility. Furthermore, Cabal has been quickly embraced by authors of layered tools mainly via the CLI. Tools which are not written in Haskell can also use the CLI to interface with Cabal.

Another way to look at Cabal's CLI is that since the packager provides a setup script, rather than just the static package description file, they have the whole power of Haskell available to them. They can use UserHooks to fill in any gaps in Cabal's behavior, they can provide their own preprocessors, or they can choose to not use the API altogether and instead build their own system, or call other tools like `autoconf` and *make*.

Therefore, the two design principals of this part of the system were first, to specify a clear and simple command-line interface, and second, to allow the packager to write a program rather than provide only static data.

4.2 Providing an API

Another design principal of Cabal was to provide an API, in the form of a Haskell library, rather than just a tool. The API is already used by Hackage and Visual Haskell to extend those tools' functionality beyond the CLI and the simple build infrastructure. Providing an API was a goal from the beginning, and we believe that many tools may benefit from providing an API rather than just a command-line tool.

4.3 Hooks and Plugins

Many tools provide hooks or a plugin architecture in order to customize their behavior. Haskell tools have a unique advantage over many systems since the hooks can actually take the form of functions, rather than a shell script, class (as in an object-orient language), or dynamic library. Cabal's hooks are very lightweight compared to such plugin architectures, and we hope that this will make it very easy to write hooks for packagers who wish to customize the behavior of Cabal's build infrastructure.

5 EXPERIENCES

The Perl programming language "was designed to be a glue language" [26] meaning that it was designed to make a wide variety of tools work together in order to accomplish system administration or system integration tasks. Python was also designed as a system administration language.

Cabal was, in part, a study in using Haskell as a "glue" language. It glues together a wide variety of tools for building software, including *GHC*, *ghc-pkg*, *nhc98*, *Hugs98*, *ld*, *autoconf*, *make*, *Greencard*, *C2hs*, *Hsc2hs*, *Alex*, *Happy*, *hscpp*, the Unix Shell, and the windows batch processing tools. It also interfaces with *cabal2rpm*, *dh_haskell*, *Hackage*, *cabal-get*, and *Visual Studio*.

Though not specifically designed as a "glue" language, I found Haskell to be very much up to the task. Haskell is a general-purpose programming language, and as such, its focus on abstraction and clarity make it good for many things, including gluing together very different kinds of tools.

In some sense, Cabal might be considered a non-idiomatic tool to implement in a purely functional language since a great deal of it operates in the IO monad. However, this was never a difficulty in Cabal, and I do not feel that the code quality suffers from being in the IO monad, nor did I feel that I lost the advantages of functional programming by operating in the IO monad. Some modules did not perform any IO at all.

Haskell's higher-order functions allow Cabal to provide the packager with a great deal of flexibility, especially in the use of user hooks. The API has already come in useful for layered tools like Hackage. Haskell's static typing helped to catch possible misunderstandings early on, and never got in the way of developing software quickly.

There are a number of standard Haskell libraries which give Haskell strength in the area of system administration. Perhaps foremost among these are `GetOpt` for parsing the command-line, the `ReadP` parser combinators for parsing the package description file, and of course the `System` hierarchy of modules for a variety of uses including file and directory manipulation, permission manipulation, and interfacing with external tools. We also made heavy use of `HUnit` for unit testing.

One drawback to Haskell for this kind of work was the lack of any standard module for constructing, indexing, and manipulating file path names. There were, however, a handful of ad-hoc path modules that had been developed over the years.

In order to fill this gap, Krasimir Angelov assembled a module, `System.FilePath`, which combined functions from Cabal's API (some of which were, in turn, taken from other tools including *GHC*), and similar modules from other languages.

In short, Haskell was a fine choice for implementing Cabal. The advantages it gives a programmer from the aspect of good engineering are very compelling compared to programming languages that are typically used for these kinds of tasks such as Perl, shell scripting languages, or even Python. Perhaps Haskell is "the best of both worlds" in that it offers the safety of static typing and the power of higher-level abstractions while at the same time providing practical, down-to-earth libraries which make it a good "glue" language.

6 RELATED WORK

6.1 Make

make is a standard Unix / Linux tool which "determines which pieces of a large program need to be recompiled, and issues commands to recompile them." [20]. A `Makefile` is similar in some ways to the combination of a Cabal package description file and Setup module, though *make* has no means of specifying package meta-data.

We chose to use Haskell rather than *make* for a variety of reasons. Its main strength is that it performs a dependency analysis and only compiles source files which require recompilation. These features are already present in *GHC*, and we

leverage them where possible. Also, *make* is not standard on all platforms, especially Windows whereas Haskell is present on all platforms we are concerned with. Also, *make* does not provide certain compelling language features like type safety and various higher-level abstractions. On balance, we felt that the dependency analysis was outweighed by these language features and portability issues. Since many Haskell tools already use *make* as a build system, Cabal provides an interface layer which can interact with *make*-based systems.

6.2 HMake

From the *HMake* web page [29], "*HMake* is an intelligent compilation management tool for Haskell programs."

HMake performs some of the same tasks as Cabal's simple build infrastructure; indeed, Cabal was very much inspired by *HMake*. Although Cabal is included with *nhc98*, it cannot build packages for *nhc98*, but one can use *nhc98* to build packages for *GHC* and *Hugs*. *HMake* can run some preprocessors, and Cabal gets some of its preprocessing code directly from *HMake*. *HMake* can also generate tracing information for Hat, the Haskell Tracer [5], performs the same kind of dependency analysis as *make*, and only rebuilds source files when necessary.

HMake was not suitable as a basis for the simple build infrastructure, however, for several reasons. It is not extensible from the packager's point of view: it cannot be used for preprocessors which it does not know about in advance, and cannot be augmented with "hooks". It also does not produce library archive files (`libHSpkgName-1.0.a`), nor can it register a library with the compilers.

Since it is designed as a tool for compiling Haskell software, rather than a packaging system, it does not provide a means to specify and parse package meta-data. It also does not provide an API, though in principal it could.

6.3 Operating System and Language Packages

Many operating systems provide a means of packaging software as described in section 1.1. Packaging systems such as dpkg, RPM, and FreeBSD's ports system [4] are language independent, but operating-system dependent. They typically provide a standard interface for building software (like the Cabal CLI), and also specify meta-data and installation instructions. Cabal is designed to work hand-in-hand with such systems on platforms where they are available.

Some other languages have a platform-independent (though language dependent) means of building and installing tools written in that language. XEmacs [10] has a packaging system which is somewhat similar to *cabal-get*; it can download and install packages, though the kind of complex processing required for Haskell packages is unnecessary for XEmacs Lisp packages.

Python's Distribution Utilities (distutils [7]) is the inspiration for Cabal's Setup module interface, and provides a function similar to Cabal's `defaultMain`. Perl

also has Cabal-like tools, including `ExtUtils::MakeMaker` [8] and `Module::Build` [21] both of which are similar to Cabal's simple build infrastructure.

Several languages have an archive of packages like Perl's CPAN [2]. CPAN is more like HackageDB than Cabal, since it is a collection of packages rather than a build system. Like Cabal, CPAN provides a means of specifying meta-data about a source file. It does not operate on collections of modules, as Cabal and Hackage do, but only on a single source file.

7 FUTURE WORK

Cabal is an evolving piece of open source software, and as such, we welcome discussion and contributions from any user community with an interest in its direction.

Cabal currently only specifies a means of distributing a single package. We hope to extend Cabal to handle collections of inter-dependent packages distributed together. We know that this would make the interaction with Visual Studio more natural, and is desirable for systems such as WASH [25] which is a collection of several packages.

Cabal currently performs no dependency analysis between modules. This seems like a natural thing to provide from the API, and may be useful for saving compilation time, as well as limiting the necessity of listing all modules explicitly in the package description file.

Cabal can already leverage the package meta-data to do more than build and install the package. For instance, it can generate Haddock documentation, as well as interact with the Programatica [17] front-end. We can take this further by interfacing with Hat (as *HMake* already does) or other debugging or code analysis tools.

Until recently, Cabal could not generate profiling libraries for *GHC* or handle mutually recursive modules. We expect that needs such as these will keep arising, as well as new preprocessors and more complex packages. We would like to add more robust support for mutually recursive modules. We hope to add *nhc98* support and support for other Haskell compilers.

There is a limitation in Haskell which disallows the use of different modules with the same name (or different versions of the same module) in a single program. This may limit some of the usefulness of Cabal and *GHC*'s new packaging system which allow different versions of the same package to be installed (provided that they are never both compiled into the same program). This has always been the case, but Since Cabal may allow more complex dependencies between packages to be practical, and since multiple versions of a package may now be installed simultaneously, we expect to see this problem arise more frequently, though it's still unclear how important it is to solve. Lifting this restriction is more properly an area of future work for Haskell compilers rather than Cabal.

Cabal requires that every package provide a Setup module, even in the case

where the module only calls `defaultMain`. This restriction may be lifted in the future, and a program provided (perhaps `cabal-setup`) in the case where a package would only use `defaultMain`. We decided to leave this restriction in place for now to simplify the explanation of Cabal's interface, and since it would be easier in the future to lift the restriction than to add it.

Future work may also include layered tools such as those outlined in section 3. HackageDB is a particularly important layered tool, since we hope it will become a large repository of Haskell packages; CPAN has been a great boon for Perl, and we would like HackageDB to do the same for Haskell.

8 CONCLUSIONS

Through use of a standard command-line interface, API, and simple build system, Cabal has provided a convenient framework for Haskell developers to get their tools into the hands of end-users. Cabal is already being used as the build system for a number of tools, including itself, other libraries in the `fptools` source tree, pesco-cmdline [16], The Haskell Cryptographic Library [24], HaXml [28], HaXR [11], and several others. It has also provided a means for layered tools such as Visual Studio and HackageDB to interact with packages.

We found Haskell to be a very fine language for the implementation of Cabal's simple build infrastructure, despite the fact that "glue" languages such as Perl or Python are more typically used for such tasks.

9 ACKNOWLEDGEMENTS

Many people have contributed significant amounts of code, patches, and bug-fixes to Cabal, especially Krasimir Angelov, Bjorn Bringert, David Himmelstrup, Simon Marlow, Ross Patterson, Martin Sjogren, and Malcolm Wallace (via *HMake*).

The original "Library Infrastructure Project" proposal was authored by Isaac Jones, Simon Marlow, Ross Patterson, Simon Peyton Jones, and Malcolm Wallace.

REFERENCES

[1] Autoconf. http://www.gnu.org/software/autoconf/.

[2] Comprehensive perl archive network. http://www.cpan.org/.

[3] Debian gnu/linux. http://www.debian.org.

[4] The freebsd ports system. http://www.freebsd.org/ports/.

[5] Hat - the haskell tracer. http://www.haskell.org/hat/.

[6] Hsc2hs. http://haskell.org/ghc/docs/latest/html/users_guide.

[7] Python distribution utilities. http://www.python.org/sigs/distutils-sig/.

[8] T. B. Andy Dougherty, Andreas Knig. Extutils::makemaker. http://perldoc.perl.org/ExtUtils/MakeMaker.html.

[9] K. Angelov. Microsoft visual studio plugin for haskell [draft], 2005.

[10] C. T. Ben Wing, Jamie Zawinski. Xemacs. Available from http://www.xemacs.org/.

[11] B. Bringert. Haxr. http://www.haskell.org/haxr/.

[12] M. M. T. Chakravarty. C2hs. http://www.cse.unsw.edu.au/ chak/haskell/c2hs/.

[13] S. M. Chris Dornan. Alex. http://haskell.org/alex/.

[14] A. Gill and S. Marlow. Happy. http://haskell.org/happy/.

[15] J. Goerzen. dh_haskell. Available with Debian GNU/Linux.

[16] S. M. Hallberg. pesco-cmdline. http://www.scannedinavian.org/ pesco/.

[17] T. Hallgren. Haskell tools from the programatica project. In *Haskell '03: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 103–106, New York, NY, USA, 2003. ACM Press.

[18] D. Herington. Hunit. http://hunit.sourceforge.net/.

[19] T. Moertel. Cabal2rpm.

[20] R. M. Richard Stallman and P. Smith. The gnu make documentation. http://www.gnu.org/software/make/manual/make.html.

[21] D. Rolsky. Module::build. http://www.perl.com/pub/a/2003/02/12/module1.html.

[22] G. N. Silva. Apt howto. http://www.debian.org/doc/manuals.

[23] T. N. Simon Peyton Jones and A. Reid. Greencard. http://www.haskell.org/greencard/.

[24] D. Steinitz. The haskell cryptographic library. http://www.haskell.org/crypto/.

[25] P. Thiemann. Web authoring system haskell. http://www.informatik.uni-freiburg.de/thiemann/.

[26] L. Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[27] M. Wallace. cpphs. http://haskell.org/cpphs/.

[28] M. Wallace. Haxml. http://haskell.org/HaXml.

[29] M. Wallace. Hmake. http://haskell.org/hmake/.

[30] D. Winer. Xml-rpc specification. http://www.xmlrpc.com/spec.