Library Infrastructure Project

Isaac Jones

The Library Infrastructure Project is an effort to provide a framework for developers to more effectively contribute their software to the Haskell community.

The Haskell Implementations[1] come included with a good set of standard libraries, but this set is constantly growing and is maintained centrally. This model does not scale up well, and as Haskell grows in acceptance, the quality and quantity of available libraries is becoming a major issue. There are a wide variety of Haskell Implementations (both compilers and interpreters), each of which target a variety of hardware architectures and operating systems. There are also a number of different groups and individuals providing libraries, and these libraries depend upon each other and they depend upon external systems such as Java or GTK.

It can be very difficult for an end user to manage these dependencies and build all the necessary software at the correct version numbers on their platform: there is currently no generic build system to abstract away differences between Haskell Implementations and operating systems [2].

The Library Infrastructure Project seeks to provide some relief to this situation by building tools to assist developers, end users, and operating system distributers.

1. High-Level Overview

On a high level, we say that we want the system to help the user to install packages (whether they be libraries or applications) and their dependencies. To accomplish this, we propose a system similar to Python's Distutils (Section A.2) where each Haskell tool is distributed with a script (Section 3.4) which has a standard command-line interface. This script will provide a familiar user interface (Section 3.3) for compiling, installing, and removing packages and their dependencies.

For instance, to install the HUnit package, the user might download the source code from the web site, change into the HUnit directory, and type **./Setup.lhs install default**, which would build and install the HUnit package for the default compiler. Similarly, he might type **./Setup.lhs install nhc** to install the package for NHC.

Other tasks might be necessary to make the package known to the system, such as registering it with the Haskell Implementation of interest (Section 4.3). Such tasks would also be performed by this Setup program.

2. Issues Facing Developers

There are a number of issues facing developers of Haskell software when they decide to deploy their work.

- Binary incompatibility between Haskell Implementations and between versions of some implementations make it very difficult to distribute binary libraries: if they are expected to work together, all of the libraries on a user's system need to be built with the same compiler. This makes it a virtual necessity that the source code for a Haskell library be distributed to the end user of that library, although other options exist for systems like Debian (Section A.1) which have auto-builders.
- Compilation is difficult: It was noted above that it is usually necessary to distribute the source code for Haskell tools to the end user. The end user has to compile each Haskell tool they use, but the task of compilation is complicated by a plethora of preprocessors (such as arrowp and greencard) and interfaces with non-Haskell libraries (such as Java, X, C, and GTK). Further, there is no way to formally express dependencies between Haskell tools, so the users have to visit the home pages of the projects they're interested in and find the dependencies.
- The Haskell Implementations are not completely compatible either: They support different extensions, different libraries, different binary formats, different flags, and different packaging systems.
- Given that compilation is difficult in and of itself, it is further complicated by a wide variety of target platforms that it is desirable to support. These systems may treat files differently (copying of files, path names) or have different defaults for where to put compiled files or source code to make them available to the user. The variety of Haskell Implementations and target platforms make writing of a generic build system that "just works" very difficult.
- Having managed to compile and install all of the Haskell software they need, if a user installs a new compiler (or version thereof), all this work needs to be redone. All of the libraries on the system will have to be recompiled to match the version of that compiler (this is apparently a bigger problem with GHC than with NHC, and not much of a problem with Hugs.)

Compare this to the experience of installing a typical library or tool written in C or C++ on a Debian GNU/Linux system: The user consults a database of known programs (which is stored on her computer) and runs a single command to install that package. The package management tool figures out the dependencies, downloads all dependencies downloads the package, and installs them in the proper order. The packages are already compiled by a set of central servers. In fact, there is no reason that this kind of interface can't be available to the end user except for the difficulty of creating binary packages. (See Section 2.1)

2.1. Issues Facing Packagers

There are roughly speaking, three different kinds of operating-system package systems: source distributions (FreeBSD, Gentoo), binary distributions (Debian, RPM), and language-specific distributions (CPAN, XEmacs). What we are proposing to create is a language-specific distribution that assists in the creation of both source and binary packages for other operating systems.

As mentioned above, it is possible to create an operating system package for a Haskell library. Indeed, its no harder to create a Haskell package than any other kind of package; Haskell packages are harder to maintain, however. For instance, in the Debian GNU/Linux system (Section A.1), which is a binary distribution, there is currently no way to express that a library needs to be recompiled when a new compiler is uploaded to the autobuilders. Currently, the only solution is to re-upload library packages to the autobuilders at the same time

you upload a new Haskell Implementation. Similarly for Redhat, binary packages have to include the version number for GHC that they were compiled with.

In a source distribution like FreeBSD (Section A.4), this isn't a problem, since the end user compiles all of their software themselves. However, when the end user compiles and installs a new Haskell Implementation, all of the libraries compiled with the old Haskell Implementation will no longer work and need to be recompiled. As I understand it, there is currently no means of performing this operation automatically.

Emacs / XEmacs presents packagers with some of the same issues; they are both, practically speaking, Lisp compilers, and incompatible, much like different versions of Haskell Implementations. When an elisp package is installed on Debian, it gets compiled automatically (on the end user's machine) for each version of Emacs / XEmacs that is installed. This procedure is orthogonal to the XEmacs Packaging System (Section A.5) and is a duplication of effort in some cases, but that is really the only way to make it work with Debian. This has some ugly properties from Debian's standpoint: once compiled, there are files in the operating system that the packaging system doesn't know about. These are similar to issues that Haskell packagers for Debian will have to face.

2.2. Why We Should Solve This

- We need to evolve a decentralized way of distributing libraries. The current model of distributing them with the Haskell Implementations puts too much strain upon the Haskell Implementation authors, and is not sustainable in the long-term. If this continues, availability and reliability of libraries will suffer, and Haskell itself will suffer in the long-term as a rich set of libraries is becoming the standard for successful programming languages.
- If we can help operating system packagers build packages (Debian, RPM, etc.), then we'll have more happy end users.
- If we lower the cost of evaluating Haskell and Haskell tools, more people might try them out.
- Many Haskell programs are developed as research prototypes and abandoned. The subsequent bitrot makes it difficult for the community to take over the projects. Formally specifying versioned dependencies and creating a central repository for Haskell tools can help solve this problem.
- Active participation with the rest of the Free Software community (inclusion of Haskell in OS distributions, interoperability between Haskell tools and external libraries) can bring attention to Haskell and perhaps bring more talented developers into The Flock.

3. A Solution for Haskell in Haskell

As mentioned above, the foremost user interface for this system will be a Haskell program to be executed by the default Haskell Implementation. This Haskell program, **Setup.lhs**, will perform the tasks of building the package (where necessary), installing the package, and making the package available to the system.

3.1. The Module Design

I propose a set of modules based on these three major tasks:

- `Distribution.Build`: For tasks of preparing the package for installation, including tasking compilers and creating packages for systems like Debian (Section 4.1).
- `Distribution.Install`: For moving the distribution files into place (Section 4.2).
- `Distribution.Package`: For accessing the database of installed packages, versions, etc. Might also be responsible for removing packages (Section 4.3).
- `Distribution.Config`: For discovering details about the configuration of the target system not covered by the database of installed packages. These are tasks typically performed by tools such as autoconf (Section 4.6).
- `Distribution`: For general purpose elements that don't fit into any of the above categories.

3.2. But Why Should We Use Haskell?

It is very appropriate that this solution be implemented in 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.

3.3. Setup.lhs Command-Line Interface

The purpose of the Setup script is to provide a standard interface to end users and layered tools. For any particular application, the script may be implemented in a variety of ways: For pure Haskell applications, the `Distribution` module should perform all of the heavy lifting, requiring only a few lines of code from the developer. For applications that feel they need a complete and robust make-based system, the Setup script can wrap such a system.

One of the early design tasks of this project should be to decide on a format for the command-line interface of the Setup script, but here is an example of how it might behave:

Table 1. Setup.lhs interface

./Setup.lhs info	Output package configuration information
./Setup.lhs build all	Compile / prepare this package for all installed Haskell Implementations

./Setup.lhs build default	Compile / prepare this package for the default Haskell Implementation
./Setup.lhs build {NHC,GHC,Hugs, ...}	Compile / prepare this package for the given Haskell Implementation
./Setup.lhs install {all,default,NHC,GHC,Hugs,...}	Install this package.
./Setup.lhs register {all,default,NHC,GHC,Hugs,...}	Register this package with the package management system (making it available to the given Haskell Implementation.)
./Setup.lhs bdist_{deb,rpm,...}	Create a binary distribution package for Debian, RPM, etc.
./Setup.lhs sdist	Create a source distribution archive.
./Setup.lhs test	Run the package's test suite.

Other commands may be available, and it is important to anticipate commands that may some day be desirable.

3.4. An Example Setup.lhs Script

Here's what the setup script might look like for HUnit, which has no complex dependencies.

```
#!/usr/bin/env haskell
import Distribution.Core
import Distribution.Package

toolInfo = (defaultPackage "HUnit"
                           (NumberedVersion 1 0 0))
   {haskellSources=[
                    "HUnitLang98.lhs","HUnitLangExc.lhs", "Info.lhs",
                    "Terminal.lhs", "HUnitTest98.lhs", "TerminalTest.lhs",
                    "HUnit.lhs", "HUnitTestBase.lhs", "HUnitBase.lhs",
                    "HUnitTestExc.lhs", "HUnitLang.lhs", "HUnitText.lhs",
                    "Setup.lhs"],
    docs = ["Example.hs", "Guide.html", "License", "README"]}

main = defaultMain toolInfo id id
-- Those last to parameters might be pre-install and post-install functions
```

`defaultMain` would implement all of the standard command-line flags, and defaultPackage is a template with sane default values for most fields.

4. Distribution Module

4.1. `Distribution.Build`

The basic strategy we will take for the actual task of building Haskell tools is as follows:

- For simple tools like Haskell modules, leverage hmake's (Section A.7) abilities and create a Haskell-based system (which will evolve to do more complex tasks.)
- Tools that require something more complex can use "fptools" or Yale's Make-based system (Section A.6), or use their own build system.
- All systems will be wrapped in a common veneer (Section 3.3) so they look the same to the average user and to layered tools (Section 6) and so that once `Distribution.Build` evolves to be a more robust tool, packages can transition to using it without effecting the interface to their build system.

Since it is obviously the compilers that do the actual compilation, the task of `Distribution.Build` is more one of coordination among tools outside the compiler. We hope to offer support for preprocessors (both existing and those yet to come). `Distribution.Build` will handle the task of compiling for a particular Haskell Implementation, or for all installed Haskell Implementations, and help to abstract-away differences between command-line flags.

`Distribution.Build` could also be used to recompile all of the installed libraries once a new Haskell Implementation is installed. This is an important function, as it solves the problem of binary incompatibility between Haskell Implementations and versions thereof. Another very useful function that `Distribution.Build` could offer is the implementation of a generic `/usr/bin/haskell` that either executes a Haskell program using the default compiler or throws the user into the default interpreter, depending upon how it is invoked. This allows Haskell scripts (such as Setup.lhs) to be distributed with a **#!/usr/bin/env haskell** annotation that has reasonable behavior.

4.2. `Distribution.Install`

The `Distribution.Install` module performs the task of moving files into place. Presumably, this is the last task before package registration. `Distribution.Install` will have to understand configuration options for the operating systems that Haskell modules are being installed on. (For instance, different operating systems have different policies for where to put documentation, source code, binary files, and libraries.) Such information will most likely be read from a file that can be edited by the system administrator (Section 4.4).

Not only will this module have to support standards on different operating systems, but it must have access to filesystem functionality like **copy** and **move**, as well as permission-related operations. Such functions should be offered by a library such as `System.Posix.Files` and `System.Directory`, but the `System.Posix` module is not available on all operating systems. To some extent, `Distribution.Install` should handle the differences between operating systems (file permissions for instance), but Haskell should offer a more robust set of file operations in order to encourage the use of Haskell for common scripting tasks. (One issue that the author has noticed is that `System.Directory.renameDirectory` is not implemented the same in GHC and Hugs, which forces `Distribution.Install` to find a way to abstract the differences.)

4.3. `Distribution.Package`

The complex task of packaging requires a lot of attention. The proposed solution is not only a module to access the packaging information, but also an application to assist external systems with the same task:

The main features of this system are:

1. To let the Haskell Implementations know how to use a package, whether its available by default (or whether it requires a -package flag), and where the root of its hierarchy is.
2. To store other information about a package, including information such as its license, home page, version number, and dependencies, to be used by other tools in the `Distribution` hierarchy.
3. All information will be made available through the `Distribution.Package` module. The information can be made available to non-haskell tools by way of a command-line tool, haskell-config (Section 4.4) with easily parsable output (similar to package-config (http://www.freedesktop.org/software/pkgconfig/)) though a different solution may be necessary for windows.

Some secondary features are:

1. Let other tools, such as debuggers and editors know where the source code for a module / package is.
2. When new Haskell implementations are installed, allow them to find the source code and import it into their own library tree (perhaps through other features of the L.I.P.)
3. For Haskell implementations that don't conform to the new packaging interface, implement a wrapper so that it can still utilize other important features of the Library Infrastructure Project.

The information would be held in a file, such as `/etc/haskell/packages.conf`[3] and `~/.haskell/packages.conf`.

4.3.1. PackageConfig Data Structure

The package data structure might look something like this (based on GHC's Package class)

```
data PkgIdentifier
    = PkgIdentifier {pkgName::String, pkgVersion::Version}
{- ^Often need name and version since multiple versions of a single
    package can exist on a system. -}

data PackageConfig
   = Package {
        pkgIdent        :: PkgIdentifier,
        license         :: License,
        auto            :: Bool,
        provides        :: [String],
{- ^A bit pi-in-the-sky; might indicate that this package provides
    functionality that other packages also provide, such as a compiler
    or GUI framework, and upon which other packages might depend. -}

        isDefault       :: Bool,
-- ^might indicate if this is the default compiler or GUI framework.

        import_dirs     :: [String],
        source_dirs     :: [String],
        library_dirs    :: [String],
        hs_libraries    :: [String],
        extra_libraries :: [String],
        include_dirs    :: [String],
        c_includes      :: [String],
        build_deps      :: [Dependency], -- build dependencies
        depends         :: [Dependency], -- use dependencies
```

```
        extra_ghc_opts  :: [String],
        extra_cc_opts   :: [String],
        extra_ld_opts   :: [String],
        framework_dirs  :: [String],
        extra_frameworks:: [String]}

data Version = DateVersion {versionYear  :: Integer,
                            versionMonth :: Month,
                            versionDay   :: Integer}
             | NumberedVersion {versionMajor      :: Integer,
                                versionMinor      :: Integer,
                                versionPatchLevel :: Integer}

data License = GPL | LGPL | BSD | {- ... | -} OtherLicense FilePath

data Dependency = Dependency String VersionRange

data VersionRange
  = AnyVersion
  | OrLaterVersion     Version
  | ExactlyThisVersion Version
  | OrEarlierVersion   Version

type PackageMap = FiniteMap PkgIdentifier PackageConfig
```

But perhaps we'll need to be even more flexible: some implementations might not be interested in certain fields, and others might want their own fields. It would certainly be desirable to have a flexible parser so that we can add more fields later and maintain backward compatibility.

The `Distribution.Package` API might look like so:

```
userPkgConfigLocation   :: FilePath
systemPkgConfigLocation :: FilePath
getSystemPkgConfig  :: IO [PackageMap] -- ^Query /etc/haskell/packages.conf
getUserPkgConfig    :: IO [PackageMap] -- ^Query ~/.haskell/packages.conf
getPkgConfig        :: FilePath -> IO [PackageMap]
addUserPackage   :: PackageConfig -> IO ()
addSystemPackage :: PackageConfig -> IO ()
delUserPackage   :: PkgIdentifier -> IO ()
delSystemPackage :: PkgIdentifier -> IO ()
basicPackage     :: PackageConfig          -- provides sensible defaults
checkLicense     :: PackageConfig -> Bool
{- Just for fun, check to see if the licences that this package uses
   conflicts with any of the licences of the packages it depends on -}
```

4.4. haskell-config Command-line interface

The haskell-config [4] tool is a command-line interface to the packaging system. It will presumably be written in Haskell and import the `Distribution.Package` module. The purpose of this tool is to give non-Haskell systems the ability to interact with the packaging system, since they won't be able to import `Distribution.Package`. This tool serves a purpose similar to ghc-pkg and package-config.

```
% haskell-config [--user] register < packageFile
% haskell-config [--user] unregister packageName
# add or remove packages from the package database.  --user indicates
# that we should add it to the package database in the user's home
# directory, not to the system-wide package database.

% haskell-config packageName c_includes
# would output this list in a way that a C compiler could use directly

% haskell-config list-packages
% haskell-config list-user-packages
% haskell-config list-system-packages
# Query the database in a variety of ways
```

4.5. haskell-pkg?

The haskell-config tool brings up an interesting question. Should the functionality of `Distribution.Install` also be made available as a command-line tool, perhaps called **haskell-pkg** ("Haskell package")? In this sense, "package" would refer to that word in the sense that dpkg and the 'P' in RPM mean it: **haskell-pkg** could be used for installing and removing Haskell programs when supplied with the package metadata that is defined by `Distribution.Package`.

4.6. `Distribution.Config`

The information available through the `Distribution.Package` module is not all of the information that could possibly be needed to prepare a package for installation. Typically, tools such as autoconf are used to discover useful information about the system. The author has not given a lot of thought to the configuration problem, but he sees a few possible paths:

- A module, `Distribution.Config` can act as an interface to configuration information. This is the approach that Python has taken.
- Some information can be written by the end user or maintained on the system in a standard file format. The file could reside, for example in `~/.haskell/distributionConfig`, `/etc/haskell/distributionConfig`, and `distributionConfig` within the package directory. This information can be made available through the `Distribution.Config` module.
- Certain pieces of information are available when the Haskell Implementations themselves are installed. This information can be made available to installing scripts once again through the `distributionConfig` file.
- Shortcomings in the `Distribution.Config` module can be made up by using autoconf itself, which can output information to the above mentioned `distributionConfig` file.

5. Use Cases

End User: The end user has identified a Haskell package (tool or library) that she wants to use.

- The end user installs packages with a operating-system-specific package management system like RPM, dpkg, or FreeBSD's Ports collection.
- If no such packaging system is available on her system, she, can run **./Setup.lhs install nhc** or **./Setup.lhs install hugs** to build, install, and register an NHC or Hugs version of the program (for instance).

Packager: A packager is someone who makes operating-system-specific packages so that an end-user can have an easier time installing them on her own system. For each platform, there should be one or more packager.

- The packager can run **./Setup.lhs bdist_deb** to build a skeleton Debian package (for instance) for each of the installed Haskell Implementations. This might generate Debian packages like *hunit-hugs*, *hunit-ghc*, *hunit-prof-ghc*, *hunit-prof-nhc*.

3rd Party Author: A 3rd party author is a Haskell developer (distinct from the Haskell Implementation authors or end users) who wishes to distribute a library or application that he has developed in Haskell.

- The 3rd Party Author writes a **Setup.lhs** program. **Setup.lhs** imports elements from the `Distribution` module which does most of the hard work. A very common case, which should be our first priority, is a pure Haskell 98 module that needn't interface with any external systems. In this case the author only has to include the name of the program, the version, and the source files. He can then call `Distribution.defaultMain` to create an executable script with the proper command-line flags that knows how to interface with the `Distribution.Package` module.
- Should he have a more complex program (one which perhaps depends on systems external to Haskell), then the `Distribution` module could output a Makefile to be used in 'fptools' or Yale's system (see Section A.6).
- Should he not want to migrate from his own build system, he could write a **Setup.lhs** script to wrap the build system so that it conforms to the standard command-line interface.

Haskell Implementation Authors:

- Haskell Implementation authors must conform to an agreed-upon `Distribution.Package` interface.
- This may include writing functionality so the compiler can be asked questions about itself.
- This may also include altering compilers to read `/etc/haskellPackages.conf` (for instance) to discover what packages are installed or where to look for imports.
- Include the `Distribution` module with the Haskell Implementations (in exchange, we can hopefully remove some libraries that are currently included with the Haskell Implementations).

Random Haskell Programmer:

- A random Haskell programmer can use the `Distribution` module as a means to create layered tools that download, build, or install packages. For instance:
  - haskell-config is an example of a layered tool which interfaces with the `Distribution` module. This tool gives the end user access to the database of installed applications.
  - A Graphical User Interface could be layered on top of the Setup script to give installation a better look-and-feel according to the target platform.
  - The `Distribution` module could be augmented with an online repository of libraries (see Section A.3), and a layered tool might perform the searching and downloading functionality of the installation process.
  - Debuggers often need to locate the source code to a library module in order to instrument them.

- Source code editors or browsers could benefit from being able to locate related source code on the user's system.

6. Means of Distribution and Layered Tools

Most of the discussion here has been about giving the user tools to build, install, and manage libraries and tools written in Haskell. There is another important component to this which deserves attention, and that is the distribution of these tools.

For C++ and Perl (and likely many other programming languages) there are central repositories of libraries and tools (Section A.3). In a way, Haskell has this also, bit it is centrally maintained rather than being a free-for-all where nearly anyone can get their package distributed. The author feels that both approaches are appropriate, and libraries can graduate from one to the other.

Having a standard interface for installing packages allows us to layer tools upon it. For instance, it would be nice to be able to download and install Haskell libraries and all their dependencies with one command. For Perl (by virtue of CPAN) this is already possible. Other layered tools are discussed in the Use Cases (Section 5).

7. Development Plan

On a high level, this is the order in which we should approach the tasks:

- We should agree first on the high-level design of the *packaging system* (Section 4.3), since this will require a consensus from the Haskell Implementation authors. Details such as the format of the database and the exact API will evolve over the course of the project. The author views this as the top priority.
- In parallel, we should decide on the command-line interface for the Setup script (Section 3.3), and the particulars of how it is invoked on various systems.
- The first priority for Build and Install support should be pure Haskell modules, and reasonable support for wrapping make-based systems that are currently more highly evolved.
- At this point, it may be possible to collect libraries, convert them to use the Library Infrastructure Project, and make them available at a central repository.
- The next priority should be tools to assist in the creation and maintenance of operating-system packages (Debian and RPM for instance).
- Once these features are in place, it makes sense to augment the `Distribution` module with more complex tool support like preprocessors and external libraries.
- Now we can add more fun features like GUI front-ends, downloading tools, package security tools, etc.

The author has implemented a "toy" prototype system that fulfills many of the features outlined above. It uses hmake by running it as an external command. It can build Debian packages and interfaces with the Common Debian Build System. It can prepare installations for Hugs and GHC. The author decided to stop implementation, however, until details about the Packaging system (Section 4.3) are worked through.

A. 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.

A.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.

A.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.

A.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.

A.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.

A.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.

A.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.

A.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. Herein, I will use Haskell Implementation as a catch-all phrase to include compilers and interpreters for the Haskell programming language.

2. Your humble author has in-depth knowledge of Debian GNU/Linux, but could probably learn a thing or two about other operating systems like Windows and MacOS. Should you notice anything here which offends your sense of operating system, please let me know.

3. Is there any Unix system where `etc` is the wrong place for something like this?

4. Because of the confusion between different kinds of configuration (the kinds offered by `Distribution.Package` and `Distribution.Config`) I am torn about the name of this program. There is the further confusion between package management (the actual installation and removal of the programs themselves) and interfacing with the packaging system. Further there is one more bit of confusion between packages in the Haskell system (i.e. a set of modules distributed together by an author) and a package on the operating system. If anyone has an idea to straighten all of this out, I'd be glad to hear it :)

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