Haskell: Batteries Included

Duncan Coutts[1] Isaac Potoczny-Jones[2] Don Stewart[2]

Well-Typed, LLP [1]
Oxford, UK
duncan@well-typed.com

Galois, Inc. [2]
Portland, Oregon
{ijones,dons}@galois.com

Abstract

The quality of a programming language itself is only one component in the ability of application writers to get the job done. Programming languages can succeed or fail based on the breadth and quality of their library collection. Over the last few years, the Haskell community has risen to the task of building the library infrastructure necessary for Haskell to succeed as a programming language suitable for writing real-world applications.

This on-going work, the *Cabal* and *Hackage* effort, is built on the open source model of distributed development, and have resulted in a flowering of development in the language with more code produced and reused now than at any point in the community's history. It is easier to obtain and use Haskell code, in a wider range of environments, than ever before.

This demonstration describes the infrastructure and process of Haskell development inside the Cabal/Hackage framework, including the build system, library dependency resolution, centralised publication, documentation and distribution, and how the code escapes outward into the wider software community.

We survey the benefits and trade-offs in a distributed, collaborative development ecosystem and look at a proposed *Haskell Platform* that envisages a complete Haskell development environment, batteries included.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]

General Terms Management, Standardization

Keywords Distribution, Functional programming

1. Introduction

Over the past twenty years a sprawling community of developers has built up around the Haskell programming language. Significant effort has gone into creating an industrial strength language, toolchain and library suite that has proven successful at tackling hard problems across a diverse range of commercial and research endeavours. (Hudak et al. 2007)

For most of its existence, Haskell has flown under the radar of the general programming community, with a famous motto of "avoid success at all costs" (Peyton Jones 2003), a phrase epitomising a culture of agility, where new research results are integrated into the language and library suite while the user base nimbly adapts. The question now is can we avoid "avoiding success". How can we best ensure a sustainable ecology of Haskell software into the future?

The key task to achieve sustainability is to provide a rich, even exhaustive, set of libraries, easily available, making Haskell a viable choice for any programming project. As a relatively small community, we aim for efficient use of developer resources, by providing:

- A standard build system
- Decentralised, lightweight collaboration resources
- A single site for publishing all libraries and tools
- Tool support for the distribution, building and installation of code on users' systems
- Groups of people to package the toolchain natively for each operating system

These systems, built by the community, go together to produce an environment that help the Haskell ecosystem thrive.

2. Infrastructure

2.1 Cabal

Cabal is a Common Architecture for Building Applications and Libraries (Jones 2005). Specifically, Cabal describes what a Haskell package is, how these packages interact with the Haskell language, and what Haskell implementations must do to support packages. Cabal also provides infrastructure code that makes it easy for authors to build and publish conforming packages. It is capable of building pure Haskell libraries and programs, as well as arbitrary mixtures of C and Haskell.

The Cabal system is largely influenced by Debian's *dpkg* tool (Debian 2003). A Cabal package consists of the source code to the application or library itself along a package description file, `package.cabal`. This largely declarative specification records what other libraries (and their versions) this package depends on, along with meta-data such as the version number and package maintainer.

This pure specification language has encouraged developers to write simple, portable software, avoiding complicated OS-specific dependencies, while encouraging the reuse of code through library packages. The specification format enables static analysis of packages, something impossible in systems such as *autoconf*, which depend on runtime configuration to determine dependencies.

2.2 Hackage

The second component is a central library and application publishing site, where all viable Haskell packages can be found, *Hackage*.

Haskell'08, September 25, 2008, Victoria, BC, Canada.
ACM 978-1-60558-064-7/08/09.

This collection is similar to Perl's *CPAN* and the *Boost* collection of libraries for C++. Package authors publish their code by uploading to Hackage. Documentation is produced by Haddock, and package dependency information is automatically generated, along with integration into central API search systems.

With straight-forward, wiki-like uploading of new packages, anyone is able to request an account and start releasing new Haskell packages to the world. By enforcing standard location and dependency information for packages, a range of tools have been implemented to process the Hackage database of packages, producing summaries, dependency analysis, and statistics about the source.

2.3 Distribution

The next piece of the puzzle is to get the source onto other developer's machines, where it can be built and used. This involves finding all required dependencies, in the correct order, and then downloading, building and installing the desired code itself. This phase is handled via *cabal-install*, inspired by Debian's *apt-get* and similar tools on other systems, which uses the Hackage database of packages to automate the build process. For example:

```
$ cabal install xmonad --dry-run
In order, the following would be installed:
mtl-1.1.0.1
X11-1.4.2
xmonad-0.7
```

This automation means interested users can get hold of new releases within minutes rather than days or months as with many native OS packaging systems. However, it does not replace native packages for end users, instead providing rapid development and beta-testing cycles for developers.

2.4 Native packages

The final link from developer to end user is via native OS packages. Using the Cabal package specification files, and central source releases on Hackage, several tools have been written to automatically produce native packages in binary and source form. Tools such as *dh_haskell*, *cabal-rpm*, *cabal2arch*, *hackport*, *mkbndl*, take a `.cabal` file as input, analyse the dependencies and build constraints, and translate the information into a OS-specific package. This may be either a binary bundle, or a source-based build specification. The process can be automated, dramatically increasing the number of packages that the native OS packaging teams can successfully manage, and ensuring Haskell packages stay up to date on end user systems.

3. The Haskell Platform

The main Haskell implementations come bundled with a common set of libraries. In the past there was great pressure to get ever more libraries into this "standard" set, mainly because external distribution was difficult. Now distribution is trivial. The impact of this reduction in cost has been to split up and reduce the bundled libraries, making it easier to upgrade or replace core components.

This has many advantages: alternative implementations of core libraries are cheaper to experiment with, and bug fixes can be delivered to users more rapidly. We harness the development power of the community more effectively by distributing package maintenance.

Unbundling is not a panacea however. Choosing between five different XML or database libraries based on incomplete information is not easy. Furthermore, there is no guarantee that an arbitrary selection of packages from Hackage is going to work together or build on a particular operating system. What is needed is a "standard", stable set of Hackage packages.

But we do not want to go back to the monolithic set of packages with slow release cycles tied to those of Haskell implementations. A more attractive project model can be seen in projects like GNOME (Miguel de Icaza 1999) or the various Linux distributions. Their solution to the problem is to select a collection of packages, and then to test and distribute them together. They have a process by which packages can be added to the collection, including various quality requirements, and they have infrastructure for testing and distribution.

3.1 A proposal

The proposal is to define a *Haskell Platform* – a collection of high quality, versioned packages from Hackage. It is intended to be a meeting point between package users, maintainers and distributors. It should provide users with a set of default libraries that implement common functionality, conveniently packaged for their operating system. For people writing packages it provides a set of dependencies which they can expect to be widely deployed and work on all major operating systems. For people providing Haskell support on their operating system it tells them what versions of what packages they need to include.

There would be regular release cycles. The initial suggestion is major releases every six months with minor bug-fix updates in between. In particular they would not be synchronised with the releases of any Haskell implementation.

The platform collection would be managed by volunteers with the help of individual package maintainers. There must be clear criteria by which new packages are accepted into the platform.

Making high quality releases of many interdependent packages on many platforms requires excellent infrastructure, as it would become an enormous time sink that no volunteer could sustain. In particular we need new features in the Hackage server to collect and summarise build and test results. Such testing and quality metrics should be available to all packages on Hackage, not just those that are part of the platform. This will help to improve quality generally, provide more information to users and make it easier for new packages to join the platform.

This proposal is not yet set in stone. It needs consensus and buy-in from package maintainers and users.

4. Results

The improvements so far have been dramatically successful. The common build system has been adopted unanimously for new Haskell projects, and dozens of libraries appear each week on Hackage. New projects are written with dependencies on previous work – code reuse is happening. The challenge now is to make the Haskell Platform ubiqitous, to produce a Haskell for the masses.

References

Debian. A Brief History of Debian. `http://www.debian.org/doc/manuals/project-history/project-history.en.tx%t`, 2003.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1 – 12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X.

Isaac Jones. The Haskell Cabal: A Common Architecture for Building Applications and Libraries. In Marko van Eekelen, editor, *6th Symposium on Trends in Functional Programming*, pages 340–354, 2005.

Miguel de Icaza. The GNOME Project: What is GNOME and where is it heading? Miguel tells us all. *Linux Journal*, page 7, 1999. ISSN 1075-3583.

Simon Peyton Jones. Wearing the hair shirt: a retrospective on Haskell (invited talk). In *ACM SIGPLAN Conference on Principles of Programming Languages (POPL'03)*, 2003.