Managing the Haskell Dependency Hell with Nix

Peter Simons

2013-04-02

Introduction

Purely functional programming languages like Haskell encourage software engineers to write re-usable code. As of today, the Haskell community's central repository "Hackage" registers 5015 packages, 3944 of which provide function libraries. This means that approximately 78% of all Haskell packages are intended to be re-used by other developers. For Haskell programmers, this is a good situation to be in! There is a lot of re-usable code out there, and the pool to choose from is growing steadily.

However, the fact that Haskell packages re-use each other a lot comes at a price: a phenomenon lovingly referred to as "dependency hell". The task of installing a Haskell package includes the installation of all its dependencies (and their respective dependencies). Updating a Haskell package typically requires updates of that package's dependencies as well. And if a package is de-installed, then chances are that some of its dependencies become unused in the process and ought to be de-installed, too.

Many Haskell programmers turn to Cabal for help — only to find that Cabal doesn't really address those tasks. Cabal can install a package, alright, but it cannot perform reliable updates or clean up the system after a de-installation. Arguably, Cabal doesn't even address the installation part satisfactorily, because it is unaware of all software outside of the domain of Haskell. The package "hsdns", for example, needs the GNU ADNS library installed to compile, but Cabal doesn't know anything about GNU ADNS. It cannot install that library as part of the hsdns build. All it can do is abort with an error message, saying "the adns library is missing", but it's the users problem to remedy that situation somehow.

Traditional package managers like `apt-get`, `rpm`, `pacman`, or `BSD ports` are designed to solve exactly those issues. These programs are really good at installing system packages. They know how to keep the installation up-to-date without breaking anything, and they know how clean-up after themselves when a package has been removed. Yet, those package managers are somewhat unprepared for the complexity of Haskell's package system. For example, most package managers assume that for any given package `A`, only one version of `A` can be installed at the same time. This is certainly not true for Haskell, where oftentimes a whole zoo of mutually incompatible library versions need to be

installed at the same time to build all installation targets. The same problem applies to the installation of the compiler. Developers who care about portable code need many different versions of GHC installed in parallel, so that their code can be tested with different compiler variants, but the ability to manage an installation of this complexity is far out of the reach of traditional package managers.

The Nix Package Manager

Nix [1] is a package manager very much like `apt-get` or `rpm`. At the same time, Nix differs from traditional package managers, because it breaks with the established Unix file system layout. Instead of installing executable programs into `/usr/bin` and libraries into `/usr/lib`, Nix installs every package into a directory hierarchy of its own. For example, all files that belong to the package GHC 7.6.2 reside underneath the prefix:

```
$ ls /nix/store/f2c7ia72ial5y8h5k1w7z5n87xnr5gm6-ghc-7.6.2/
bin  lib  share
```

This installation scheme allows any number of GHC versions to co-exist, because they live in completely different places:

```
$ ls /nix/store/*-ghc-*/bin/ghc
/nix/store/cczdxjhrx27wmf4rpk4rk70zabw2fggd-ghc-6.10.4/bin/ghc
/nix/store/n6ygpkh29dfp9di27l70681k7llnhama-ghc-6.12.3/bin/ghc
/nix/store/frziw92yvb0rgm7sx7zzsmgy7cyl16d9-ghc-7.0.4/bin/ghc
/nix/store/82lkfs5c4l41p244apv7q5dmd91h712d-ghc-7.4.2/bin/ghc
/nix/store/ch0cq5l0l8p7g80skifivrg4d40s8sdm-ghc-7.6.2/bin/ghc
```

Furthermore, packages compiled by Nix generally don't refer to any system path outside of `/nix/store`. GHC depends on system libraries like glibc, libgmp, and curses, but all those dependencies are satisfied entirely within Nix:

```
$ ldd ghc
/nix/store/ml128hlikbl14q4cwviafpaskskx1a3f-ncurses-5.9/lib/libncursesw.so.5
/nix/store/zpr8jdx10napidkyz62f3nzl5fm87jyr-glibc-2.17/lib/librt.so.1
/nix/store/zpr8jdx10napidkyz62f3nzl5fm87jyr-glibc-2.17/lib/libutil.so.1
/nix/store/zpr8jdx10napidkyz62f3nzl5fm87jyr-glibc-2.17/lib/libdl.so.2
/nix/store/zpr8jdx10napidkyz62f3nzl5fm87jyr-glibc-2.17/lib/libm.so.6
/nix/store/zpr8jdx10napidkyz62f3nzl5fm87jyr-glibc-2.17/lib/libpthread.so.0
/nix/store/zpr8jdx10napidkyz62f3nzl5fm87jyr-glibc-2.17/lib/libc.so.6
/nix/store/989njbbf770sxyqqpyrw2m0cbfdynzbb-gmp-5.0.5/lib/libgmp.so.10
```

The `/nix/store` hierarchy is completely self-contained. A Haskell development environment that's been created on one machine can be copied to any other machine simply by running "`rsync -a /nix other-machine:/`".

A quick glance at the store paths shown above reveals yet another powerful feature. The exact path used to store an installed package contains a cryptographic hash that uniquely identifies

1. the shell command sequence that built this particular package and
2. all other store paths that this package depends on.

Suppose that GHC 7.6.2 would be compiled two times. The first build uses GCC version 4.6.3 to build the Haskell compiler, but the second build uses GCC 4.7. Then the results of those two builds would be stored in paths like

- `/nix/store/ch0cq5l0l8p7g80skifivrg4d40s8sdm-ghc-7.6.2` and
- `/nix/store/f2c7ia72ial5y8h5k1w7z5n87xnr5gm6-ghc-7.6.2`.

The cryptographic hashes assigned to those two paths differs because both variants of GHC 7.6.2 have different dependencies (GCC 4.6.3 and 4.7 respectively). Because of this approach, Nix is able to satisfy the requirements of the Haskell package system, which needs to keep multiple versions of the same package around concurrently. Nix can easily do that, simply by having a separate store path for each variant of the package.

These capabilities make Nix rather well suited for maintaining sophisticated Haskell development environments, and the remainder of this article is intended to provide a hands-on tutorial of how to use Nix for exactly that purpose.

Installing the Haskell Platform

The installation procedure of Nix itself is fairly straightforward. Ready-to-use binary packages are available for many Linux platforms (particularly on x86, x86_64, and PowerPC), for Mac OS X, and for FreeBSD. Furthermore, Nix can be compiled from source code on any POSIX-like operating system. Detailed installation instructions can be found at [2].

Once Nix itself is installed, it can be used to obtain an up-to-date copy of the Nix package database by running:

```
$ nix-channel --add http://nixos.org/channels/nixpkgs-unstable
$ nix-channel --update
```

Now, it is possible to see the entire set of packages known to Nix by running:

```
$ nix-env -qaP '*'
pkgs.pong3d               3dpong-0.5.0
pkgs.rdf4store            4store-v1.1.5
pkgs.foursuite            4suite-1.0.2
...                       ...
```

The output consists of two columns. The right hand-side shows the human-readable name and version of the package. The left hand-side shows a unique identifier for that particular package in the Nix package database. Both identifiers can be used to install packages:

```
$ nix-env -i 3dpong-0.5.0     # install by name
$ nix-env -iA pkgs.pong3d     # install by attribute
```

Now, to install the Haskell Platform, we first need to make up our mind which version we want, because there are plenty to choose from. The command

```
$ nix-env -qaP '*' | grep haskell-platform
```

shows that all of the versions 2009.2.0.2, 2010.1.0.0, 2010.2.0.0, 2011.2.0.0, 2011.2.0.1, 2011.4.0.0, 2012.2.0.0, and 2012.4.0.0 are available, and each of those versions of Haskell Platform can be compiled with GHC versions 6.10.4, 6.12.1, 6.12.2, 6.12.3, 7.0.1, 7.0.2, 7.0.3, 7.0.4, 7.2.1, 7.2.2, 7.4.1, 7.4.2, 7.6.1, and 7.6.2 respectively!

For the sake of the example, let's just choose the latest official version:

```
$ nix-env -iA haskellPackages.haskellPlatform
installing `haskell-haskell-platform-ghc7.4.2-2012.4.0.0'
...
building path(s) `/nix/store/j1gq9xciqj3390himzi3p9xcln9hnxl4-user-environment'
created 1159 symlinks in user environment
```

When that command is run for the first time, it will download and install a *lot* of packages, because the Haskell Platform depends on many other things such as the Linux kernel headers, glibc, GCC, GMP, Perl, and so on. Of course, these packages will be re-used when installing a different version of the Haskell Platform later, so one doesn't have to worry that every single install is going to be that expensive.

In its last step, `nix-env` creates a "user environment" at `$HOME/.nix-profile`, which simplies use of the installed packages a lot. That environment uses symlinks to create a view of the Nix store that combines those packages that have been installed into a single hierarchy. For example, the path `~/.nix-profile/bin/ghc` will be a symlink to `/nix/store/vlx3ikjjxlfvqjjlx74cg07p79y43mdq-ghc-7.4.2/bin/ghc`.
The path `~/.nix-profile/bin/alex`, on the other hand, points to `/nix/store/7rfilvshj9wzm3rx86m4mkpf76l24fhs-alex-3.0.2/bin/alex`.
The binaries `alex` and `ghc` live in different store paths, but the user environment joins all those paths into a single hierarchy. This means that it is sufficient to add `$HOME/.nix-profile/bin` to `$PATH` to use the Haskell Platform from Nix.

References

[1] Nix, the purely functional package manager: http://nixos.org/nix/

[2] Nix Installation Instructions: http://hydra.nixos.org/build/4480483/download/1/manual/#chap-installation