Conquer Cabal Hell with Nix

@steshaw

Overview of talk

1. Purely Functional Package Systems
2. Overview of Nix
3. How to use Nix with Haskell

Purely Functional Package Systems

Features

- multiple variants of a package at the same time (i.e. side by side)
- immutable packages
- atomic upgrades/rollbacks
- mutiple environments
- usable as non-root

Examples

- Nix
- Zero Install
- Listaller (was Autopackage)

Nix

- Source-based with “binary replacement”.
- Scales up to entire Linux distribution. See NixOS.
- Written in C++ and Perl but looks like a Haskell port is in the works.
- Popular in the Haskell community.

Commands

Find packages

```
$ nix-env --query --available 'hello*'
$ nix-env -qa 'hello*'
```

Install a package

```
$ nix-env --install hello
$ nix-env -i hello
```

Remove a package

Notes (work in progress)

Nix Features

- Add more Nix features from `http://nixos.org/nix/about.html`
- Diagrams!

Architecture of Nix

Under the hood a little bit

- /nix/store
- Mention the crazy linker thing.
- How hashing works. Why binary substiution works.
- A -> B. What are the inputs?
  - What inputs?
  - What outputs?
  - hashing for the store
- Caching/Memoisation.

Nix Commands

Accelerating Haskell Development with Nix

Where are we now?

- Using cabal sandboxes.
- Perhaps some shared sandboxes.
- Waiting for builds is no fun.
- Wasting time building ‘lens‘ for each of your projects that uses it is not good.
- Let’s not accept the status quo.
- One option is to use Halcyon – a build cache for Cabal.

What’s not good?

- Long build times.
- Building the same dependencies over and over again in different sandboxes.
- These sandboxes could be on your machine or your team members machine.
- Or on the build box.
- There is wastage of time but also of disk space.