Nix-ify your Psc-Package dependencies

Justin Woo

Helsinki Haskell

2018 Dec 04

What is Nix again?

The parts of Nix that we care about in this talk:

- `Nix` - the package manager that works via derivations of packages and stores of built packages
- `Nixlang` - the programming language used by the package manager to build derivations

What we don't really care about in this talk:

- `NixOS` - the Linux distribution that uses all of the parts of Nix

What is Psc-Package?

- A package manager for PureScript based on Package Sets
- Works via Git repositories
- Basically only does three things: Solves for dependencies, Install dependencies, Build dependencies + src/
- “What is solving” -> more on that later, but basically finding all of the transitive dependencies

What are Package Sets

- A... set of packages (aka a record of packages): fields of package names and definitions

```
{
  // ...                 // other things like foreign, etc
  "simple-json": {       // what is the name of this package?
    "dependencies": [    // what does this package depend on?
      "record",
      "exceptions",      // IMPORTANT:
      "foreign",         // all of these only refer to a dependency by *name*
      "foreign-object",  // all packages only have a *single* version per set
      "globals",         // there is no such thing as having multiple versions
      "nullable",        // this is how it's a set
      "variant"
    ],
    "repo": "https://github.com/justinwoo/purescript-simple-json.git",
    "version": "v4.4.0" // tag name
  },
  // ...                 // everything else
}
```

Problem

Using Psc-Package normally leads to a whole lot of waste:

- The dependency is specified as `.psc-package/${SET_NAME}/${PACKAGE_NAME}/${PACKAGE_VERSION}/`
- If a dependency doesn't exist at the expected directory, it is git cloned again
- Packages are downloaded again even at the same version, since they are not of the same package set

"Why doesn't Psc-Package solve this?"

- I'm the main maintainer now for Psc-Package
- I hate buggy features, especially naive caching ("try clearing the npm/stack/cabal/apt/bundler/cargo/yum/pacman/... cache!!!")
- Let's not reinvent the wheel in a poor way when there's already tool that can do this correctly

Psc-Package Objectives

- I should be able to use a regular old `psc-package.json` config:

```
{
  "name": "test",  // unimportant project name
  "set": "241018", // which tag of my package set i am using
  "source": "https://github.com/justinwoo/spacchetti.git",
                   // ^ the url of my package set
  "depends": [     // my direct dependencies
    "aff",
    "console"
  ]
}
```

- I should solve the transitive dependencies of my project and generate a set of derivations for my packages using the package set
- These derivations should only be based on their versions, so that different package sets still generate the same derivations, so that Nix reuses them across package sets
- I should have some way to copy these files into the correctly structure in `.psc-package/` so that psc-package can use them seamlessly

Derivations

- This is basically a recipe of a input source to an output which will be stored in the... store

While this is also a *set* (aka record) with some basic properties of name, system (e.g. x86_64-linux), and builder (binary program to build the derivation), in practice we will almost never make a raw derivation:

```
derivation {
  name = "myname";
  builder = "mybuilder";
  system = "mysystem";
}
```

Instead, we will almost always work with the nixpkgs `mkDerivation` function:

```
{ pkgs ? import <nixpkgs> {} }: # allow an argument `pkgs` into this expression
                                # but use nixpkgs as the default
{
  # ...
  DEP = pkgs.stdenv.mkDerivation {
    name = NAME;                # name of the derivation
    version = VERSION;          # some metadata we cram in here
    src = pkgs.fetchgit {       # the fetch from git function in nixpkgs
      url = REPO;               # repo link of package from package sets
      rev = REV;                # the revision/version of the package
      sha256 = HASH;            # the sha256 hash of the git repo
    };
    dontInstall = true;         # don't run the installation phase
    buildPhase = ''             # build the derivation by copying its contents into the output
      cp --no-preserve=mode,ownership,timestamp -r $src $out
    '';
  };
  # ...
}
```

Parameters

In our derivation template, we need to supply the following:

- DEP - for our attribute name
- NAME - for the derivation name
- VERSION - for metadata
- REPO - for the git repo link
- REV - for the revision to be used
- HASH - to verify the input contents

These can be supplied quite simply from the package set information from before

```
"simple-json": {       // used for both DEP and NAME
  "dependencies":      // don't care here
  "repo": "https://github.com/justinwoo/purescript-simple-json.git",
                       // ^ REPO
  "version": "v4.4.0"  // VERSION and REV
},
```

HASH:

```
nix-prefetch-git $repo --rev $version --quiet
```

e.g.

```
> nix-prefetch-git https://github.com/justinwoo/spacchetti.git 241018 --quiet
{
  "url": "https://github.com/justinwoo/spacchetti.git",
  "rev": "49e083d0884669ac5d4a9a74e466bd711c4020ae",
  "date": "2018-10-23T13:43:03+03:00",
  "sha256": "0k7iv0i65d1l5rshh682b8ayszd3n02cwa3g32p2fg0mz37llaip",
  "fetchSubmodules": false
}
```

Now we know how to get all of the parameters we need!

Transitive dependencies

We need to figure out what all packages to get, which we will do by visiting all declared dependencies of our dependencies.

```
# called by a parent getDeps subroutine
sub getDepsInner {
    my ($name, $json, $visited) = @_;
    chomp($name);

    if ($visited->{$name}) { return; }

    $visited->{$name} = 1;
    my @transitive_deps = `jq '.${name}.dependencies | values[]' $json`;
    foreach my $target (@transitive_deps) {
        getDepsInner($target, $json, $visited);
    }
    return;
}
```

That's it!

Packages.nix

So in the end we can produce an output file with the package set information and all of the individual derivations:

```
{ pkgs ? import <nixpkgs> {} }:

let inputs = {DERIVATIONS}; # each derivation is independent of the package set
in {
  inherit inputs;
  set = SET;                # but set and source information are still available to query
  source = SOURCE;          # because we need this to make the .psc-package structure
}
```

So while we might update package sets in the future, any input derivation that retains the same version and repository will have the same input derivation hash, so we can use the stored outputs in the Nix store.

Consumption

The way I want to consume these files is by copying them into my dependencies, so I can freely edit them as desired.

This also means that I will want to copy them once over per installation and not constantly, so I essentially just want some shell hooks to be run.

The easy way to do this:

```
nix-shell install-deps.nix --run 'installation complete'
```

install-deps.nix

```
let
  pkgs = import <nixpkgs> {};

  packages = import ./packages.nix {};               # pull in the generated packages
  packageDrvs = builtins.attrValues packages.inputs; # get the actual derivations from the set

  pp2n-utils = import pkgs.fetchurl {                # utils.nix from psc-package2nix
    url = "...";
    sha = "...";
  };

in pkgs.stdenv.mkDerivation {
  name = "install-deps";
  shellHook = pp2n-utils.mkDefaultShellHook packages packageDrvs;
                                                     # ^ make a shell hook using utils that copies our deps
}
```

utils.nix

```
rec {
  mkCopyHook = packages: drv:
  let target = ".psc-package/${packages.set}/${drv.name}/${drv.version}";
  in ''
    if [ ! -e ${target} ]; then
    mkdir -p ${target}
    cp --no-preserve=mode,ownership,timestamp -r ${toString drv.outPath}/* ${target}
    fi
  '';

  mkDefaultShellHook = packages: drvs: toString (map (mkCopyHook packages) drvs);
}
```

Usage

First, prepare `default.nix` with things we'll need:

```
let
  pkgs = import <nixpkgs> {};

  easy-ps = import (pkgs.fetchFromGitHub {
    owner = "justinwoo";
    repo = "easy-purescript-nix";
    rev = "dac3520da91bf1b2d152d468700b75be5599b784";
    sha256 = "02lcmsscbq1k3c8ap03xxbrf4vbwi1al6hsvfsr3sry7xj8f7ca4";
  });

in pkgs.stdenv.mkDerivation {
  name = "test";
  buildInputs = [
    pkgs.jq
    pkgs.nix-prefetch-git
    easy-ps.inputs.purs
    easy-ps.inputs.psc-package-simple
    easy-ps.inputs.psc-package2nix
  ];
}
```

Then use everything via nix-shell, e.g. a Makefile:

```
default:
    nix-shell --run 'make build'
build:
    psc-package2nix
    nix-shell install-deps.nix --run 'echo installation complete'
    psc-package build
```

```
> make
nix-shell --run 'make build'
make[1]: Entering directory '/home/justin/Code/psc-package2nix/test'
psc-package2nix
# Cloning into '.psc-package/241018/.set'...
# fetching .psc-package2nix/...
wrote packages.nix
nix-shell install-deps.nix --run 'echo installation complete'
installation complete
psc-package build
# Compiling ...
make[1]: Leaving directory '/home/justin/Code/psc-package2nix/test'
```

That's it!

Psc-Package2Nix summary

- Psc-Package2Nix automatically prepares derivations for our dependencies specified in the regular Psc-Package config
- every time we change package sets, only packages that have changed will have to be redownloaded
- we can choose when we want to actually run the dependency installation process, no need to always deal with `nix-build` or whatever
- implemented in Perl 5 + jq + git + nix-prefetch-git, so anyone can contribute and easily run this on their machine, even without using Nix
- easy to bring in via Nix via fetchFromGitHub/Easy-PureScript-Nix/etc

Thanks

Some links:

- Psc-Package2Nix https://github.com/justinwoo/psc-package2nix
- Easy-PureScript-Nix https://github.com/justinwoo/easy-purescript-nix