

# Stumbling around with Nix and Haskell development

Dave Laing

# Introduction

# What is Nix?

- ▶ Purely functional package manager

# What is Nix?

- ▶ Reliable (through not overwriting things)

# What is Nix?

- ▶ Reproducible

# What is Nix?

- ▶ Lots and lots of scope for caching

# What is this talk about?

There is plenty of information around about how Nix works and how to use it.

# What is this talk about?

Some of the Haskell related information is out of date or incomplete.



What is this talk about?

Some of it is fine.

What is this talk about?

It can be hard to tell the difference.

# What is this talk about?

The aim of this talk is to give you a cheatsheet so you can do a handful of common things while reading about / experimenting with the rest of what Nix has to offer.

## What is this talk about?

You can do a lot of stuff for global usage in your user profile.

## What is this talk about?

If you come across people talking about `~/.nixpkgs/config.nix`, that's a good sign they are talking about the setup of the user profile.

# What is this talk about?

Most of this talk will be about working with self-contained environments.

## What is this talk about?

If you come across people talking about `shell.nix`, that's a good sign they are talking about the setup of a self-contained environment.

Getting started



## Installing nix

```
sudo mkdir /nix  
sudo chown dave /nix  
bash <(curl https://nixos.org/nix/install)
```

## Removing nix

```
rm -rf /nix
```

## Installing nix via reflex-platform

```
git clone https://github.com/reflex-frp/reflex-platform  
cd reflex-platform  
./try-reflex
```

## Installing cabal2nix

We'll install this using nix:

```
nix-env -i cabal2nix
```

## Installing cabal2nix

This installs it in the user profile, so that it's available all of the time.

# Installing cabal2nix

We can update it with:

```
nix-env -u cabal2nix
```

## Installing cabal2nix

We can erase it with:

```
nix-env -e cabal2nix
```

Files



```
default.nix
```

```
cat my-project.cabal
```

## default.nix

```
name:                my-project
version:             0.1.0.0
license:             BSD3
license-file:       LICENSE
...
library
  exposed-modules:    ...
  build-depends:      base                >= 4.8   && < 4.9
                     , containers          >= 0.5   && < 0.6
                     , lens                >= 4.13  && < 4.14
                     , mtl                 >= 2.2   && < 2.3
  hs-source-dirs:     src
  ghc-options:        -Wall
  default-language:   Haskell2010
```

default.nix

We use cabal2nix to get going with nix:

```
cabal2nix . > default.nix
```

## default.nix

This is a function, which produces a derivation:

```
{ mkDerivation, base, containers, lens, mtl
, stdenv
}:
mkDerivation {
  pname = "my-project";
  version = "0.1.0.0";
  src = ./.;
  libraryHaskellDepends = [
    base containers lens mtl
  ];
  license = stdenv.lib.licenses.bsd3;
}
```

default.nix

This is a building block, since the dependencies aren't concrete at this point.

default.nix

It is in a form where we can reuse it in various contexts.

shell.nix

```
cabal2nix --shell . > shell.nix
```

## shell.nix

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "default" }:  
let  
  inherit (nixpkgs) pkgs;  
  
  f = ... what we saw in default.nix ...  
  
  haskellPackages =  
    if compiler == "default"  
    then pkgs.haskellPackages  
    else pkgs.haskell.packages.${compiler};  
  
  drv = haskellPackages.callPackage f {};  
in  
  if pkgs.lib.inNixShell then drv.env else drv
```



shell.nix

We can set options and pick version here.

shell.nix

Now the dependencies all have defaults.

shell.nix

This is in a form where we can use it immediately.

default.nix and shell.nix

We can do both.

## default.nix and shell.nix

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "default" }:
let
  inherit (nixpkgs) pkgs;

  haskellPackages =
    if compiler == "default"
    then pkgs.haskellPackages
    else pkgs.haskell.packages.${compiler};

  -- imports default.nix if no file name is
  -- mentioned explicitly
  drv = haskellPackages.callPackage ./ . {};
in
  if pkgs.lib.inNixShell then drv.env else drv
```

## default.nix and shell.nix

Now we can do

```
cabal2nix . > default.nix
```

whenever we change our .cabal file, and we don't need to update  
shell.nix

## default.nix and shell.nix

We can use `shell.nix` to do things straight away, and we can use `default.nix` to slice and dice things as we need to.

# Commands



## nix-shell

If we have a `shell.nix` file, we can do

```
nix-shell
```

to drop into a development environment with all of the dependencies that we asked for.

## nix-shell

We still have access to all of the tools and libraries outside of the nix environment.

## nix-shell

We can use:

```
nix-shell --pure
```

to remove the access to things that we haven't mentioned.

## nix-build

We can build the artifact described by `shell.nix` using

```
nix-build shell.nix
```

## `nix-build`

The result will be in `./result`, and a Nix GC root will be created for the result.

`nix-build`

This is what your CI system will typically run.

# Haskell development

We can do our Haskell development from outside or inside of the nix shell.

## Haskell development - from the outside

In the same directory as `shell.nix`:

```
nix-shell --command 'cabal configure'
```



## Haskell development - from the outside

`cabal configure` caches absolute paths, so we can do other cabal commands after this without having to be inside the nix shell.

## Haskell development - from the inside

```
dave> nix-shell  
nix-shell> emacs
```

(or vim, or cabal repl, or ...)

## Dependencies

## Non-haskell dependencies

```
"sdl2" = callPackage
  ({ mkDerivation, base, bytestring, exceptions, linear
    ,           , StateVar, text, transformers, vector
  }):
mkDerivation {
  pname = "sdl2";
  version = "2.1.3";
  sha256 = "ce18963594fa21d658deb90d22e48cd17e499b23...";
  libraryHaskellDepends = [
    base bytestring exceptions linear StateVar
    text transformers vector
  ];

  license = stdenv.lib.licenses.bsd3;
}) { }
```

## Non-haskell dependencies

```
"sdl2" = callPackage
  ({ mkDerivation, base, bytestring, exceptions, linear
    , SDL2, StateVar, text, transformers, vector
    }:
    mkDerivation {
      pname = "sdl2";
      version = "2.1.3";
      sha256 = "ce18963594fa21d658deb90d22e48cd17e499b23...";
      libraryHaskellDepends = [
        base bytestring exceptions linear StateVar
        text transformers vector
      ];

      license = stdenv.lib.licenses.bsd3;
    }) {
```

## Non-haskell dependencies

```
"sdl2" = callPackage
  ({ mkDerivation, base, bytestring, exceptions, linear
    , SDL2, StateVar, text, transformers, vector
    }:
    mkDerivation {
      pname = "sdl2";
      version = "2.1.3";
      sha256 = "ce18963594fa21d658deb90d22e48cd17e499b23...";
      libraryHaskellDepends = [
        base bytestring exceptions linear StateVar
        text transformers vector
      ];

      license = stdenv.lib.licenses.bsd3;
    }) {inherit (pkgs) SDL2};
```

## Non-haskell dependencies

```
"sdl2" = callPackage
  ({ mkDerivation, base, bytestring, exceptions, linear
    , SDL2, StateVar, text, transformers, vector
    }:
    mkDerivation {
      pname = "sdl2";
      version = "2.1.3";
      sha256 = "ce18963594fa21d658deb90d22e48cd17e499b23...";
      libraryHaskellDepends = [
        base bytestring exceptions linear StateVar
        text transformers vector
      ];
      librarySystemDepends = [ SDL2 ];

      license = stdenv.lib.licenses.bsd3;
    }) {inherit (pkgs) SDL2};
```

## Non-haskell dependencies

```
"sdl2" = callPackage
  ({ mkDerivation, base, bytestring, exceptions, linear
    , SDL2, StateVar, text, transformers, vector
    }:
    mkDerivation {
      pname = "sdl2";
      version = "2.1.3";
      sha256 = "ce18963594fa21d658deb90d22e48cd17e499b23...";
      libraryHaskellDepends = [
        base bytestring exceptions linear StateVar
        text transformers vector
      ];
      librarySystemDepends = [ SDL2 ];
      libraryPkgconfigDepends = [ SDL2 ];
      license = stdenv.lib.licenses.bsd3;
    }) {inherit (pkgs) SDL2};
```



## Local Haskell dependencies

```
{ mkDerivation, base, containers, lens, mtl
, stdenv
}:
mkDerivation {
  pname = "my-project";
  version = "0.1.0.0";
  src = ./.;
  libraryHaskellDepends = [
    base containers lens mtl

  ];
  license = stdenv.lib.licenses.bsd3;
}
```

## Local Haskell dependencies

```
{ mkDerivation, base, containers, lens, mtl
, stdenv, my-dependency
}:
mkDerivation {
  pname = "my-project";
  version = "0.1.0.0";
  src = ./.;
  libraryHaskellDepends = [
    base containers lens mtl

  ];
  license = stdenv.lib.licenses.bsd3;
}
```

## Local Haskell dependencies

```
{ mkDerivation, base, containers, lens, mtl
, stdenv, my-dependency
}:
mkDerivation {
  pname = "my-project";
  version = "0.1.0.0";
  src = ./.;
  libraryHaskellDepends = [
    base containers lens mtl
    my-dependency
  ];
  license = stdenv.lib.licenses.bsd3;
}
```

## Local Haskell dependencies

```
modifiedHaskellPackages = haskellPackages.override {  
  overrides = self: super: {  
  
    my-project =  
      self.callPackage ./.. {};  
  };  
};  
  
drv = modifiedHaskellPackages.my-project;
```

## Local Haskell dependencies

```
modifiedHaskellPackages = haskellPackages.override {  
  overrides = self: super: {  
    my-dependency =  
      self.callPackage ../my-dependency {};  
    my-project =  
      self.callPackage ../. {};  
  };  
};  
  
drv = modifiedHaskellPackages.my-project;
```

## Pinning versions

```
modifiedHaskellPackages = haskellPackages.override {  
  overrides = self: super: {  
    # The default version of mtl will get  
    # passed along to the derivation function in  
    # default.nix  
  
    my-project =  
      self.callPackage ../. {};  
  };  
};  
  
drv = modifiedHaskellPackages.my-project;
```

## Pinning versions

```
modifiedHaskellPackages = haskellPackages.override {  
  overrides = self: super: {  
    # We can grab a particular version from Hackage
```

```
    my-project =  
      self.callPackage ./ . {};  
  };  
};
```

```
drv = modifiedHaskellPackages.my-project;
```

## Pinning versions

```
modifiedHaskellPackages = haskellPackages.override {  
  overrides = self: super: {  
    # We can grab a particular version from Hackage  
  
    mtl =  
      self.callHackage "mtl" "2.2.1" {};  
    my-project =  
      self.callPackage ./.. {};  
  };  
};  
  
drv = modifiedHaskellPackages.my-project;
```



Git dependencies

## The manual way

- ▶ clone the repository from git
- ▶ run cabal2nix in the repository
- ▶ add the repository as a dependency as above

## The manual way

This is pretty poor for reproducibility - we should keep better track of our sources.

## Using cabal2nix

You can specify a package on Hackage:

```
cabal2nix cabal://mtl > mtl.nix
```

## Using cabal2nix

```
{ mkDerivation, base, stdenv, transformers }:  
mkDerivation {  
  pname = "mtl";  
  version = "2.2.1";  
  sha256 = "1icdbj2rshzn0m1zz5wa7v3xvkf6qw811p4s7jgqvw...";  
  revision = "1";  
  editedCabalFile = "4b5a800fe9edf168fc7ae48c7a3fc2aab...";  
  libraryHaskellDepends = [ base transformers ];  
  homepage = "http://github.com/ekmett/mtl";  
  license = stdenv.lib.licenses.bsd3;  
}
```

## Using cabal2nix

You can specify a version:

```
cabal2nix cabal://mtl-2.2.1 > mtl.nix
```

which is the same in this case.

## Using cabal2nix

You can even work straight from git:

```
cabal2nix https://github.com/ekmett/mtl > mtl.nix
```

## Using cabal2nix

```
{ mkDerivation, base, fetchgit, stdenv, transformers }:  
mkDerivation {  
  pname = "mtl";  
  version = "2.2.2";  
  src = fetchgit {  
    url = "https://github.com/ekmett/mtl";  
    sha256 = "032s8g8j4dix7y3f8ryfmg6rwsmxhzhxha2qh1fj...";  
    rev = "f75228f7a750a74f2ffd75bfbf7239d1525a87fe";  
  };  
  libraryHaskellDepends = [ base transformers ];  
  homepage = "http://github.com/ekmett/mtl";  
  license = stdenv.lib.licenses.bsd3;  
}
```



## Using nix-prefetch-scripts

```
nix-env -i nix-prefetch-scripts
```

## Using nix-prefetch-scripts

The scripts fetch files into the nix store, and give us the information that we need in order to refer to those files unambiguously.

## Using nix-prefetch-scripts

```
> nix-prefetch-git https://github.com/ekmett/mtl
...
{
  "url": "https://github.com/ekmett/mtl",
  "rev": "f75228f7a750a74f2ffd75bfbf7239d1525a87fe",
  "date": "2016-09-28T05:55:27-04:00",
  "sha256": "032s8g8j4dix7y3f8ryfmg6rwsmxhzhxha2qh1fj..."
}
```

## Using nix-prefetch-scripts

Output from the scripts:

```
{  
  "url": "https://github.com/ekmett/mtl",  
  "rev": "f75228f7a750a74f2ffd75bfbf7239d1525a87fe",  
  "date": "2016-09-28T05:55:27-04:00",  
  "sha256": "032s8g8j4djsx7y3f8ryfmg6rwsmxhzhxha2qh1fj..."  
}
```

Use in mtl.nix:

```
...  
src = fetchgit {  
  url = "https://github.com/ekmett/mtl";  
  sha256 = "032s8g8j4djsx7y3f8ryfmg6rwsmxhzhxha2qh1fj..."  
  rev = "f75228f7a750a74f2ffd75bfbf7239d1525a87fe";  
};  
...
```

# Using nix-prefetch-scripts

Problem: `fetchGit` clones the whole repository.

## Using nix-prefetch-scripts

```
...  
src = fetchgit {  
  url = "https://github.com/ekmett/mtl";  
  
  sha256 = "032s8g8j4dix7y3f8ryfmg6rwsmxhzhxha2qh1fj...  
  rev = "f75228f7a750a74f2ffd75bfbf7239d1525a87fe";  
};  
...
```

## Using nix-prefetch-scripts

```
...  
src = fetchFromGitHub {  
  url = "https://github.com/ekmett/mtl";  
  
  sha256 = "032s8g8j4dix7y3f8ryfmg6rwsmxhzhxha2qh1fj...  
  rev = "f75228f7a750a74f2ffd75bfbf7239d1525a87fe";  
};  
...
```

## Using nix-prefetch-scripts

```
...
src = fetchFromGitHub {
  url = "https://github.com/ekmett/mtl";
  repo = "mtl";
  sha256 = "032s8g8j4dix7y3f8ryfmg6rwsmxhzhxha2qh1fj...
  rev = "f75228f7a750a74f2ffd75bfbf7239d1525a87fe";
};
...
```



## Using nix-prefetch-scripts

```
...  
src = fetchFromGitHub {  
  owner = "ekmett";  
  repo = "mtl";  
  sha256 = "032s8g8j4djsx7y3f8ryfmg6rwsmxhzhxha2qh1fj...  
  rev = "f75228f7a750a74f2ffd75bfbf7239d1525a87fe";  
};  
...
```

## Using nix-prefetch-scripts

Uses the GitHub API to download a .zip file of just the revision we're after.

## Using nix-prefetch-scripts

The SHA is done on the contents, which is why we can reuse the info from `nix-prefetch-git`.

## Automating cabal2nix

These techniques works if the repository already has a `default.nix`.

## Automating cabal2nix

If we use cabal2nix on a git repository to get a default.nix, we have to carry around nix information for all of our dependencies.

## Automating cabal2nix

We can automate the use of `cabal2nix`, since `nix` is a *language*.

## Automating cabal2nix

We add a function to run cabal2nix:

```
cabal2nixResult =
  src: nixpkgs.runCommand "cabal2nixResult" {
    buildCommand = ''
      cabal2nix file://"${src}" >"$out"
    '';
    buildInputs = with nixpkgs; [
      cabal2nix
    ];
  } "";
```

(which we got from reflex-platform)

## Automating cabal2nix

We add data for our sources:

```
sources = {  
  mtl = pkgs.fetchFromGitHub {  
    owner = "ekmett";  
    repo = "mtl";  
    sha256 = "032s8g8j4dix7y3f8ryfmg6rwsmxhzhxha2qh1fj...";  
    rev = "f75228f7a750a74f2ffd75bfbf7239d1525a87fe";  
  };  
};
```



## Automating cabal2nix

And then we combine those two pieces:

```
modifiedHaskellPackages = haskellPackages.override {  
  overrides = self: super: {  
    mtl =  
      self.callPackage (cabal2nixResult sources.mtl) {};  
    my-project =  
      self.callPackage ././ {};  
  };  
};
```

## Tweaking things with `lib.nix`

There is a file in `nixpkgs` that provides some handy functions for tweaking derivations.

## Tweaking things with lib.nix

...

```
doHaddock = drv: overrideCabal drv  
  (drv: { doHaddock = true; });
```

```
dontHaddock = drv: overrideCabal drv  
  (drv: { doHaddock = false; });
```

```
doJailbreak = drv: overrideCabal drv  
  (drv: { jailbreak = true; });
```

```
dontJailbreak = drv: overrideCabal drv  
  (drv: { jailbreak = false; });
```

```
doCheck = drv: overrideCabal drv  
  (drv: { doCheck = true; });
```

```
dontCheck = drv: overrideCabal drv  
  (drv: { doCheck = false; });
```

...

## Tweaking things with lib.nix

```
modifiedHaskellPackages = haskellPackages.override {  
  overrides = self: super: {  
    mtl =  
  
      self.callPackage (  
        cabal2nixResult sources.mtl  
      ) {};  
  
    my-project =  
      self.callPackage ./ . {};  
  };  
};
```

## Tweaking things with lib.nix

```
inherit (nixpkgs) pkgs;
lib =
  import "..snip../haskell-modules/lib.nix" {
    pkgs = nixpkgs;
  };

modifiedHaskellPackages = haskellPackages.override {
  overrides = self: super: {
    mtl =

      self.callPackage (
        cabal2nixResult sources.mtl
      ) {};

    my-project =
      self.callPackage ./.. {};
  };
};
```

## Tweaking things with lib.nix

```
inherit (nixpkgs) pkgs;
lib =
  import "..snip../haskell-modules/lib.nix" {
    pkgs = nixpkgs;
  };

modifiedHaskellPackages = haskellPackages.override {
  overrides = self: super: {
    mtl =
      lib.dontCheck (
        self.callPackage (
          cabal2nixResult sources.mtl
        ) {}
      );
    my-project =
      self.callPackage ./.. {};
  };
};
```

# Options

## Choosing a compiler

We already have support for choosing a compiler in our `shell.nix` file.



## Choosing a compiler

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "default" }:  
let  
  inherit (nixpkgs) pkgs;  
  
  f = ... what we saw in default.nix ...  
  
  haskellPackages =  
    if compiler == "default"  
    then pkgs.haskellPackages  
    else pkgs.haskell.packages.${compiler};  
  
  drv = haskellPackages.callPackage ./ . {};  
in  
  if pkgs.lib.inNixShell then drv.env else drv
```

## Choosing a compiler

We could lock this down.

## Choosing a compiler

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "default" }:  
let  
  inherit (nixpkgs) pkgs;  
  
  haskellPackages =  
    if compiler == "default"  
    then pkgs.haskellPackages  
    else pkgs.haskell.packages.${compiler};  
  
  drv = haskellPackages.callPackage ./ . {};  
in  
  if pkgs.lib.inNixShell then drv.env else drv
```

## Choosing a compiler

```
{ nixpkgs ? import <nixpkgs> {},                                }:  
let  
  inherit (nixpkgs) pkgs;  
  
  haskellPackages =  
    if compiler == "default"  
    then pkgs.haskellPackages  
    else pkgs.haskell.packages.${compiler};  
  
  drv = haskellPackages.callPackage ./ . {};  
in  
  if pkgs.lib.inNixShell then drv.env else drv
```

## Choosing a compiler

```
{ nixpkgs ? import <nixpkgs> {},                                }:  
let  
  inherit (nixpkgs) pkgs;  
  
  haskellPackages =  
    pkgs.haskell.packages.ghc7103;  
  
  drv = haskellPackages.callPackage ./ . {};  
in  
  if pkgs.lib.inNixShell then drv.env else drv
```

## Choosing a compiler

We could choose the compiler from the command line.

## Choosing a compiler

From `shell.nix` we can see the options:

```
{ nixpkgs ? import <nixpkgs> {}, compiler ? "default" }:  
...
```

which we can set while calling `nix-shell`:

```
nix-shell --arg compiler ghc7103
```

## Adding profiling

```
...
    modifiedHaskellPackages = haskellPackages.override {
        overrides = self: super: {

            my-project = self.callPackage ./ . {} ;
        };
    };
...
```



## Adding profiling

```
...
modifiedHaskellPackages = haskellPackages.override {
  overrides = self: super: {
    mkDerivation = args: super.mkDerivation (args // {
      enableLibraryProfiling = true;
    });
    my-project = self.callPackage ../. {};
  };
};
...
```

## Adding hoogle support

```
...
  modifiedHaskellPackages = haskellPackages.override {
    overrides = self: super: {

      my-project = self.callPackage ./ . {};
    };
  };
...
```

## Adding hoogle support

```
...
modifiedHaskellPackages = haskellPackages.override {
  overrides = self: super: {
    ghc = super.ghc // {
      withPackages = super.ghc.withHoogle;
    };
    ghcWithPackages = self.ghc.withPackages;
    my-project = self.callPackage ./.. {};
  };
};
...
```

## Conclusion

## Worth poking about in

- ▶ `nixpkgs`
- ▶ `reflex-platform`

## Other places to look for information

The Haskell section of the `nixpkgs` manual

- ▶ <https://nixos.org/nixpkgs/manual>

is really good, and has some info on Stack integration as well.

## Other places to look for information

This page

- ▶ <http://www.cse.chalmers.se/~bernardy/nix.html>

is also really good.

## Other places to look for information

Ollie Charles has a blog post related to this

- ▶ <https://ocharles.org.uk/blog/posts/2014-02-04-how-i-develop-with-nixos.html>

but his wiki entry

- ▶ <http://wiki.ocharles.org.uk/Nix>

seems to be more up-to-date.