Simplifying Unstructured Grids for Oceanographic Visualizations

Ole Tytlandsvik

December 6, 2024

The “Solution”

Functions:

```
{ inputs = { ... }; }
```

- Dependencies are inputs
- Usually tarballs or git repos
- Pinned and hashed

```
{ outputs = inputs: { ... }; }
```

- Outputs are functions of inputs
- Can be anything
- Lazily evaluated

What is Nix?

- Just a programming language
- Functional
  - lazy
  - everything is an expression
- Turing complete
- Made to configure environments
  - native paths
  - tooling for environments -> nixos etc

Trinity

- Nix: the package manager
- NixDSL: the programming language
- Nixpkgs: the repository
- NixOS: the operating system

Nix REPL

The Nix REPL (Read-Eval-Print Loop) is an interactive environment for evaluating Nix expressions.

```
nix repl -f '<nixpkgs>'
```

Useful commands:

- `:l <path>`: load a file
- `:q`: quit
- `:t`: show type of expression
- `:t <expr>`: show type of expression

Language Basics

Integers:

```
> x = 1 + 1
> x
2
```

Floats:

```
> y = 1.0 + 1.0
> y
2.0
```

Strings:

```
> z = "world"
> "hello ${z}"
"hello world"
```

Attribute sets:

```
> s = { a = { b = 1; }; }
> s.a.b
1
```

Language Basics

Lists:

```
> [ 1 "2" (_: 3) ]
[ 1 "2" <thunk> ]
```

Recursive attrsets:

```
> rec { x = 1; y = x; }
{ x = 1; y = 1; }
```

Bindings:

```
> let x = 1; in x + 1
2
```

Inherits:

```
> let x = 1; y = x; in
    { inherit x y; }
{ x = 1; y = 1; }
```

Language Basics

Functions 1:

```
> f = x: x + 1
> f 2
3
> g = g': x: g' x + 1
> g f 2
4
```

Functions 2:

```
> h = { x ? 1 }: x + 1
> h
<function>
> h { }
2
> h { x = 2; }
3
```

Derivation

A *derivation*

- is a plan / blueprint
- it's used for producing
  - `lib`: library outputs
  - `bin`: binary outputs
  - `dev`: header files, etc.
  - `man`: man page entries
  - ...

```
derivation ::
  { system    : String
  , name      : String
  , builder   : Path | Drv
  , ? args    : [String]
  , ? outputs : [String]
  } -> Drv
```

Derivation

Example:

```
derivation ::
  { system    : String
  , name      : String
  , builder   : Path | Drv
  , ? args    : [String]
  , ? outputs : [String]
  } -> Drv
```

```
derivation {
  system = "aarch64-linux";
  name = "hi";
  builder = "/bin/sh";
  args = ["-c" "echo hi >$out"];
  outputs = ["out"];
}
```

Derivation

Special *variables*:

```
derivation {
  system = "aarch64-linux";
  name = "hi";
  builder = "/bin/sh";
  args = ["-c" "echo hi >$out"];
  outputs = ["out"];     ^^^^
}              ^^^
```

- `$src`: build source
- `$out`: build output (default)
- custom outputs

Nix Store

```
/nix/store/l2h1lyz50rz6z2c8jbni9daxjs39wmn3-hi
|---------|--------------------------------|--|
store     hash                             name
prefix
```

- Store prefix can be either local or remote (`binary cache`)
- Hash either derived from input (`default`) or output (`CA derivation`)
- The hash ensures two realised derivations with the same name have different paths if the inputs differ at all

Packaging

The process of: Nix expressions ⇒ derivation(s)

- `builtins.derivation`
- `stdenv.mkDerivation` (from `nixpkgs`)
- `pkgs.buildDotnetModule` (from `nixpkgs`)
- …

Packaging[1]

```
{ stdenv
, lib
, pkgs
}:
pkgs.writeShellApplication {
    name = "moo";
    version = "0.0.1";
    runtimeInputs = [ pkgs.cowsay ];
    text = "cowsay moo";
}
```

```
 _____
< moo >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

[1]Example 1

Development[1]

Shell:

- `nix develop`
- `direnv`

```
pkgs.mkShell {
  nativeBuildInputs = with pkgs; [
    cargo
    rustc
    rustfmt
  ];
};
```

Formatter:

- `nixfmt`
- a single package, or ↓

```
formatter = pkgs.writeShellScriptBin "formatter" ''
  set -eoux pipefail
  shopt -s globstar
  ${pkgs.nixpkgs-fmt}/bin/nixpkgs-fmt .
  ${pkgs.rustfmt}/bin/rustfmt **/*.rs
'';
```

[1] Example 2

Pinning

w/ builtin versions:

```
nix-repl> pkgs.coq_8_
pkgs.coq_8_10  pkgs.coq_8_12
pkgs.coq_8_14  pkgs.coq_8_16
pkgs.coq_8_18  pkgs.coq_8_5
pkgs.coq_8_7   pkgs.coq_8_9
...
```

w/ `nix shell`:

```
nix shell nixpkgs/<hash>#{pkg1,...}
```

or DIY![1]

w/ npins or niv:

```
let
  sources = import ./nix;
  pkgs = import sources.nixpkgs {};
in {
  # Use pinned packages
  hello = pkgs.hello;
}
```

Initialize npins with:

```
npins init
# Channel
npins add channel 24.05
# GitHub
npins add github cachix/git-hooks.nix
```

[1] https://github.com/andir/npins

System Configuration

i.e. NixOS

- A GNU/Linux distribution
- Fundamentally different file system design
  - nix store
  - otherwise just like any penguin variant
- Only configures and installs system wide programs
  - use home-manager for user-based configuration

System Configuration

```
outputs = { nixpkgs, ... }: {
  nixosConfigurations."coolpc" = nixpkgs.lib.nixosSystem {
    specialArgs = {
      inherit inputs pkgs;
    };
    modules = [ /* A list of modules goes here */ ];
  };
};
```

System Closure:

```
nix build -f . nixosConfigurations.coolpc.config.system.build.toplevel
```

Rebuild:

```
nixos-rebuild <switch|boot|...>
```

What else is Nix good for?

- CI/CD
  - declarative and reproducible pipelines
  - no version mismatch due to nix
  - available as a github runner -> nix-run
- Kubernetes
  - declarative/reproducible deployments
  - easily convertible to and from docker files/images

Challenge

- Create hello world apps in whatever language you want
- Package it in nix
  - `default.nix`
  - direnv with `shell.nix`
- Have everyone else make them work by just entering your folder
- Try adding dependencies
- Starter is provided in the `starter` folder

Resources

- Installer: https://nix.dev/install-nix
- REPL is your friend: `nix repl`
- Intro: https://nix.dev/tutorials/first-steps/
- Manual: https://nixos.org/manual/nix/unstable/
- Forum: https://discourse.nixos.org
- Options: https://mynixos.com
- Source code search:
  - https://github.com/features/code-search