Nix modules: Improving Nix's discoverability and usability

Eelco Dolstra
Tweag Software Innovation Lab
eelco.dolstra@tweag.io

2020-10-16

Goal: Make Nix more user-friendly

Goal: Make Nix more user-friendly

Warning: Vaporware ahead

Overview

Problems:

- Nix is not very beginner-friendly — steep learning curve

Overview

Problems:

- Nix is not very beginner-friendly — steep learning curve
- Nix/Nixpkgs has lots of configuration mechanisms but most are...
  - Not easily discoverable
  - Not uniform
  - Not easy to use

Overview

Problems:

- Nix is not very beginner-friendly — steep learning curve
- Nix/Nixpkgs has lots of configuration mechanisms but most are...
  - Not easily discoverable
  - Not uniform
  - Not easy to use

Solutions:

- Provide a uniform, discoverable configuration mechanism

Overview

Problems:

- Nix is not very beginner-friendly — steep learning curve
- Nix/Nixpkgs has lots of configuration mechanisms but most are...
  - Not easily discoverable
  - Not uniform
  - Not easy to use

Solutions:

- Provide a uniform, discoverable configuration mechanism
- Make things discoverable/configurable via the Nix CLI

Overview

Problems:

- Nix is not very beginner-friendly — steep learning curve
- Nix/Nixpkgs has lots of configuration mechanisms but most are...
  - Not easily discoverable
  - Not uniform
  - Not easy to use

Solutions:

- Provide a uniform, discoverable configuration mechanism
- Make things discoverable/configurable via the Nix CLI
- Provide a simpler configuration language (TOML) for simple projects

Configuration mechanisms in Nixpkgs

- Function arguments
- `.override`
- `.overrideDerivation`
- `config`
- Overlays
- NixOS modules

Function arguments

pkgs/applications/misc/blender/default.nix:

```
{ cudaSupport ? false
, cudatoolkit
}:

stdenv.mkDerivation {
  name = "blender-2.90.1";
  buildInputs = [ boost ... ]
    ++ (if cudaSupport then [ cudatoolkit ] else []);
  ...
}
```

Function arguments (cont'd)

Problems:

- Not discoverable: only way to find out about `cudaSupport` is to read the source
- Cannot be overriden from the command line:

  ```
  nix-build -A blender --arg cudaSupport true
  ```

  doesn't work
- No type checking
- Mixes user-configuration options (`cudaSupport`) with dependencies (`cudatoolkit`)

`.override` / `.overrideDerivation`

```
my-blender = blender.override {
  cudaSupport = true;
};
```

Problems:

- Inefficient: Calls the function *twice*
- Leaks memory

config

`~/.config/nixpkgs/config.nix`:

```
{ cudaSupport = true; }
```

`pkgs/applications/misc/blender/default.nix`:

```
{ config,
, cudaSupport ? config.cudaSupport or false
, cudatoolkit
}: ...
```

Problems:

- Not discoverable
- No type checking
- Not obvious to use from the CLI

Overlays

```
nixpkgs.overlays = [ (self: super: {
  hostapd = super.hostapd.overrideDerivation (prev: {
    patches = prev.patches or []
      ++ [ ./no-scan.patch ];
  });
}) ];
```

- Not obvious how nested overrides work
  (e.g. in `pythonPackages` or `perlPackages`)

NixOS modules

- Discoverable
- Options have documentation
- Type checking
- Not well integrated into Nix

Solution: Nix modules

Let's turn (something like) NixOS modules into a language feature and use it everywhere!

A Hello World module

```
modules.hello = module {
  doc = "A program that prints a friendly greeting.";
  extends = [ nixpkgs.modules.package nixpkgs.modules.stdenv ];
  options = {
    who = {
      default = "World";
      doc = "Who to greet.";
    };
  };
  config = { config }: {
    pname = "hello";
    version = "1.12";
    buildCommand =
      ''
        mkdir -p $out/bin
        cat > $out/bin/hello <<EOF
        #! /bin/sh
        echo Hello ${config.who}
        EOF
        chmod +x $out/bin/hello
      '';
  };
};
```

Using a module

```
$ nix run .#hello
Hello World
```

Using a module

```
$ nix run .#hello
Hello World
```

which is equivalent to

```
$ nix run .#modules.hello.final.derivation
Hello World
```

Discoverability

```
$ nix flake show
git+file:///path/to/flake?dir=hello
|---modules
    |---hello

$ nix list-options .#hello
who
  Description: Who to greet.
  Value: "World"
...
```

Overriding module options from the CLI

```
$ nix run .#hello --argstr who NixCon
Hello NixCon
```

Overriding a module from another flake

```
{
  inputs.hello.url = github:tweag/nix-ux?dir=hello;

  outputs = { self, hello }: {

    modules.my-hello = module {
      extends = [ hello.modules.hello ];
      config = { config }: {
        who = "NixCon";
      };
    };

  };
}
```

Documentation!

```
$ nix doc /path/to/flake
$ xdg-open ./flake-doc/index.html
```

`derivation`

Synopsis

A Nix derivation.

Options

- `buildCommand`

  The contents of the shell script that builds the derivation.

- `derivation`

  The resulting derivation.

- `environment`

  Environment variables passed to the builder.

- `name`

Making Nix beginner-friendly: TOML flakes

```
nix.toml:

[inputs]
hello.url = "github:tweag/nix-ux?dir=hello"

[my-hello]
extends = [ "hello#hello" ]
who = "NixCon"
```

Changing flakes from the CLI

```
$ nix set-option .#my-hello who 'NixCon 2020'

$ nix run .#my-hello
warning: Git tree '/home/eelco/Dev/nix-ux' is dirty
Hello NixCon 2020

$ git commit -a -m 'Changed some options'
```

Conclusion

Next steps

- Better syntax/semantics for modules
- More experiments

More information

- PoC implementation:
  `https://github.com/NixOS/nix/tree/configs`
- Some examples:
  `https://github.com/tweag/nix-ux/`
- More language ideas:
  `https://gist.github.com/edolstra/29ce9d8ea399b703a7023073b0dbc00d`

Package generators as modules

- `modules.derivation`
  - Wrapper around `builtins.derivation`
  - Input options `name`, `system`, `environment`, `buildCommand`
  - Output option `derivation`

Package generators as modules

- `modules.derivation`
  - Wrapper around `builtins.derivation`
  - Input options `name`, `system`, `environment`, `buildCommand`
  - Output option `derivation`
- `modules.stdenv`
  - Extends `modules.derivation`
  - Input options `modules.buildInputs`, ...
  - Sets `modules.buildCommand`

Package generators as modules

- `modules.derivation`
  - Wrapper around `builtins.derivation`
  - Input options `name`, `system`, `environment`, `buildCommand`
  - Output option `derivation`
- `modules.stdenv`
  - Extends `modules.derivation`
  - Input options `modules.buildInputs`, ...
  - Sets `modules.buildCommand`
- `modules.package`
  - Extends `modules.derivation`
  - Input options `modules.pname`, `modules.version`,
  - Sets `modules.name`

Package generators as modules

- `modules.derivation`
  - Wrapper around `builtins.derivation`
  - Input options `name`, `system`, `environment`, `buildCommand`
  - Output option `derivation`
- `modules.stdenv`
  - Extends `modules.derivation`
  - Input options `modules.buildInputs`, ...
  - Sets `modules.buildCommand`
- `modules.package`
  - Extends `modules.derivation`
  - Input options `modules.pname`, `modules.version`,
  - Sets `modules.name`
- `modules.pythonPackage`, `modules.perlPackage`, ...

Example

```
modules.package = module {
  doc = "An installable package.";

  extends = [ self.modules.derivation ];

  options = {

    pname = {
      example = "hello";
      doc = "Name of the package.";
    };

    version = {
      default = null;
      example = "1.2.3";
      doc = "The version of the package. Must be null or start with a digit.";
    };

  };

  config = { config }: {
    name = config.pname +
      (if config.version != null then "-" + config.version else "");
  };
};
```