Units of Composition

recipes, overlays, and packages

Thomas Bereknyei (tomberek) flox

Introduction

- What is this called?
- Why it matters?
- What problems we encounter?
- Proposals
- Examples
- References

Disclaimer: Intermediate Nix experience helpful

What is this called?

```
{ stdenv , fetchurl }:

stdenv.mkDerivation (finalAttrs: {
  pname = "hello";
  version = "2.12.1";

  src = fetchurl {
    url = "mirror://gnu/hello/hello-${finalAttrs.version}.tar.gz";
    sha256 = "sha256-jZkUKv2SV28wsM18tCqNxoCZmLxdYH2Idh9RLibH2yA=";
  };
})
```

. . .

Often called a “package”, but that’s not quite right?

The main idea of this talk is to explain how we work with this, and to suggest we give it a name.

Docker has a name for the image, and names for containers, not the recipe. It might produce a package. It is missing

Package: take 1

> Create a **package**.nix file in the **package** directory, containing a Nix expression — a piece of code that describes how to build the **package**. In this case, it should be a function that is called with the **package** dependencies as arguments, and returns a build of the **package** in the Nix store.

Nixpkgs pkgs/README.md

Package: take 2

> Nix doesn't really have a notion of "package". The term is only mentioned in a few places in the code, ... Nixpkgs on the other hand is all about packages, but it does not define precisely what a package is.

Nix Issue #6507

roberth proposed a definition of package

Package: take 3

> I think we need to expose all the functions we callPackage on their own.
>
> As a middle ground, also expose the function to be fixed ("all packages") but no fixed point "yet"

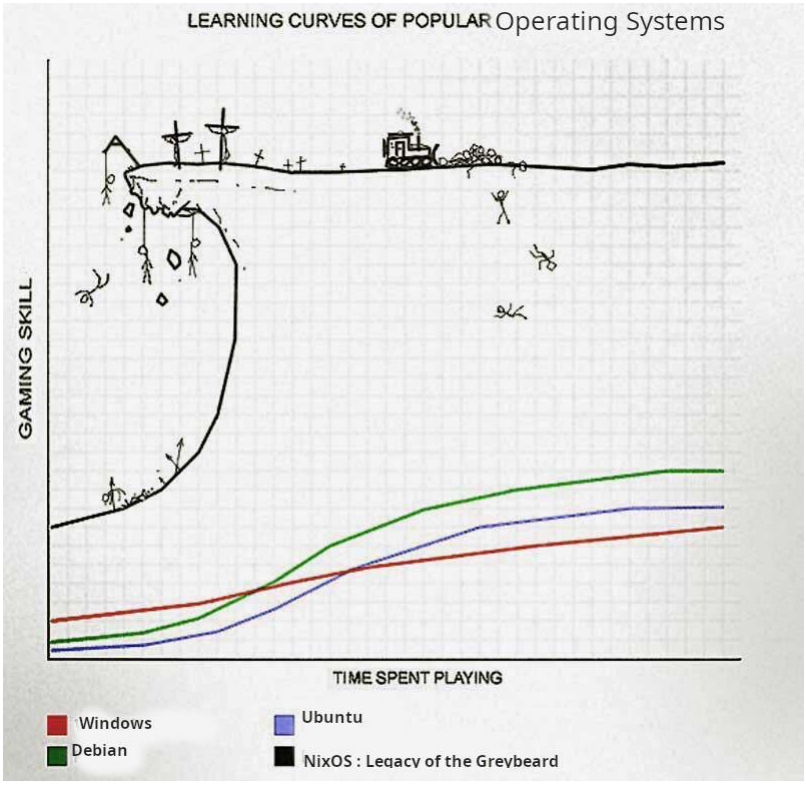
Nixpkgs Issue #172008

The value of a name

- They allow us to communicate.
- They allow us to teach.
- They allow us to precisely define abstractions.

why we should care

- This **thing** is used throughout Nixpkgs
- Beginners encounter this.
- We build further abstractions over it.
- Nix should understandable.



There is a long history of the importance of having a name. Knowing a name gives you power over it.

problems

- "I created a package. How can I build it?"
- "I got a package to build, how can i add it to Nixpkgs?"
- "My other packages can't see my own package."
- "My NixOS/home-manager can't see my package."
- "What is an overlay?"
- Overlays, fixed points, callPackage: oh my!
- "What is a flake? How do I add my package?"

callPackage

- A function which will call your definition with the correct arguments from a *scope*[1] and provide a few usability benefits such as overrides.

[1] https://github.com/NixOS/nixpkgs/blob/master/lib/customisation.nix#L308

- Used throughout Nixpkgs to avoid tedious and error-prone threading of dependencies from their declaration to where they are used.
- good reference at: https://summer.nixos.org/blog/callpackage-a-tool-for-the-lazy/

poorly named

callPackage: overview

```
let
  callPackageWith = scope: f: extra:
    let argsFrom = builtins.intersectAttrs (builtins.functionArgs f);
    f (argsFrom scope // extra);

  callPackage = callPackageWith ({
    a = 1;
    b = 2;
  } // packages);

  packages = {
    c = callPackage ({a}: a + 2) {};
    d = callPackage ({a,c}: a + c) {};
  };
in
  packages
```

define the helper

```
  # define a function with three arguments
  callPackageWith = scope: f: extra:

    let argsFrom =
      # extract those arguments from the scope
      builtins.intersectAttrs
        # extract the required arguments of the function
        (builtins.functionArgs f);

    # call the original function with the extracted args
    f (argsFrom scope // extra);
```

define callPackage

```
  callPackageWith = scope: f: extra: ...;

  # "capture" a scope that remaining callers have access to
  callPackage = callPackageWith (
```

```
    # a simple scope (or Nixpkgs)
    {
        a = 1;
        b = 2;
    }
```

. . .

```
    # The most mind-boggling thing.
    # Expand the scope with the packages we are about to define.
    # Requires lazy language.
    // packages );
```

callPackage captures a closure and extends it

using callPackage

```
{
  callPackageWith = scope: f: extra: ...;
  callPackage =            f: extra: ... // packages);

  packages = {
    c = callPackage functionC {};
    d = callPackage functionD {};
  };
}
```

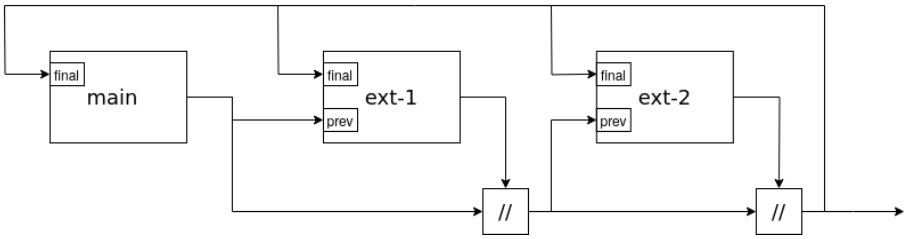
This looks reasonable. Next, one would want to make this set of extensions available and re-usable, we've given this concept a name: "overlays".

using overlays

```
  callPackageWith = scope: f: extra: ...;
  callPackage =            f: extra: ...;
  packages    = {...};

  overlay = final: prev: {
    c = final.callPackage functionC {};
    d = final.callPackage functionD {};
  };
```

What is final? prev? Does anyone understand this?



overlays

- overlays are very powerful
- error prone: infinite recursion, nested sets, ...
- most users don't need that full expressivity
- most common to add a packge or two to the scope
- difficult to extract the original re-usable function

. . .

Overlays are the **correct** way compose packages, but are hard to use.

Package sets

```
callPackageWith = scope: f: extra: ...;
callPackage =            f: extra: ...;

packages    =
   # Provide the base packages and the new ones.
   let pkgs = {...};
   in
     # Include hooks to be able to further add more.
     pkgs // { inherit callPackage extend pkgs; };
```

Package set features

- We have several in Nixpkgs, but not standardized
  - pkgs (top-level)
  - pythonPackages + python3Packages
  - haskellPackages
  - perlPackages
  - ...
- Includes the machinery needed to use.
- Relatively unknown how they work.
- Difficult to nest: try overriding in pythonPackages

NAT Proposal: standardize + document package sets

scopes

Not a full treatment of the topic, but worth being aware of. Creating a scope allows one to add a bunch of packages to a set, compose everything, then later extract only the ones you added.

```
callPackageWith = scope: f: extra: {...};

makeScope = callPackageWith: f:
  let self = f self // {
    callPackageWith = scope: callPackageWith (self // scope);
    packages = f;
  };
in self;
```

nixpkgs internals

Proposals

Things we can discuss and do today.

name this thing

```
{ stdenv , fetchurl }:

stdenv.mkDerivation (finalAttrs: {
  pname = "hello";
  version = "2.12.1";

  src = fetchurl {
    url = "mirror://gnu/hello/hello-${finalAttrs.version}.tar.gz";
    sha256 = "sha256-jZkUKv2SV28wsM18tCqNxoCZmLxdYH2Idh9RLibH2yA=";
  };
})
```

Proposal: Names

- package: related, but misses key concepts
- package function: correct, but awkward
- derivation: not until resolved
- proto-derivation: correct, but awkward
- blueprint: sterile
- recipe: instructions which allow variations

Any name is better than no name?

recipe

- instructions
- allows for variations
- cookbooks

standard flake output

```
recipes = {
  my-app-a = import ./pkgs/my-app-a/;
  my-app-b = {runCommand}: runCommand "b" {} "touch $out";
  my-app-c = {hello}: hello.overrideAttrs (_: {name = "c";});
  my-data = {}: "some data, some data";
};
```

. . .

- no “system”, friendly to cross-compiling
- obvious translation from a “cookbook” into overlays
- “recipes” as a an official top-level flake output.
- nixpkgs expose them prior to being callPackage’d.
- no lockfiles needed
- frameworks: FUP, flake-parts, devenv, flox, etc.

no lockfile bloat

```
recipes.packages = {
  my-app-a = import ./pkgs/my-app-a/;
  my-app-b = {runCommand}: runCommand "b" {} "touch $out";
  my-app-c = {hello}: hello.overrideAttrs (_: {name = "c";});
  my-data = {}: "some data, some data";
};
```

These are pure functions with no references to a system or a nixpkgs. So they can be accessed without needing to bring in transitive inputs.

additional thoughts

```
{stdenv, fetchurl}:        # User question: "what am I allowed to put here?"

stdenv.mkDerivation {
  pname = "bbbb";
  version = "1.0";
  src = ...;
}
```

Hard question to answer if someone has used overlays, overrides, added new packages, or are in a nested package set. We can expose this scope directly!

```
$ nix search .#context gcc
$ nix search .#scope.myPackages gcc
```

What is next?

- no underlying techincal changes required
- a social convention is enough to start
- thoughts?
- RFC?
- add support in libraries and frameworks
- developer experience needs to expand

"using"

```
using baseNixpkgs {

  hello-go = ./pkgs/hello-go;
  hello-perl = ./pkgs/hello-perl;

  python3Packages = {
    hello-python-library = ./pkgs/python3Packages/hello-python-library;
  };
  hello-python = ./pkgs/hello-python;
```

```
  # Escape-hatch into full nixpkgs overrides
  hello-python-override =
      callPackage: (callPackage ./pkgs/hello-python {})
         .overrideAttrs (_: {name="hello-python-override";});
}
```

Demo?

No time, but this approach exists in various forms.

This talk about trying to explain and then change how we think about such topics.

References

Nixpkgs pkgs/README.md Nix Issue #6507 Nixpkgs Issue #172008 customisation https://summer.nixos.org/blog/callpackage-a-tool-for-the-lazy/ nixpkgs internals