Nix Pills

Luca Bruno

Preface

This is a ported version of the **Nix Pills**, a series of blog posts written by **Luca Bruno** (aka Lethalman) and originally published in 2014 and 2015. It provides a tutorial introduction into the Nix package manager and Nixpkgs package collection, in the form of short chapters called 'pills'.

Since the Nix Pills are considered a classic introduction to Nix, an effort to port them to the current format was led by Graham Christensen (aka grahamc / gchristensen) and other contributors in 2017.

For an up–to–date version, please visit https://nixos.org/guides/nix–pills/. An EPUB version is also available.

If you encounter problems, please report them on the nixos/nix–pills issue tracker.

Why You Should Give it a Try

Introduction

Welcome to the first post of the "Nix in pills" series. Nix is a purely functional package manager and deployment system for POSIX.

There's a lot of documentation that describes what Nix, NixOS and related projects are. But the purpose of this post is to convince you to give Nix a try. Installing NixOS is not required, but sometimes I may refer to NixOS as a real world example of Nix usage for building a whole operating system.

Rationale for this series

The Nix, Nixpkgs, and NixOS manuals along with the wiki are excellent resources for explaining how Nix/NixOS works, how you can use it, and how cool things are being done with it. However, at the beginning you may feel that some of the magic which happens behind the scenes is hard to grasp.

This series aims to complement the existing explanations from the more formal documents.

The following is a description of Nix. Just as with pills, I'll try to be as short as possible.

Not being purely functional

Most, if not all, widely used package managers (dpkg, rpm, ...) mutate the global state of the system. If a package `foo-1.0` installs a program to `/usr/bin/foo`, you cannot install `foo-1.1` as well, unless you change the installation paths or the binary name. But changing the binary names means breaking users of that binary.

There are some attempts to mitigate this problem. Debian, for example, partially solves the problem with the alternatives system.

So while in theory it's possible with some current systems to install multiple versions of the same package, in practice it's very painful.

Let's say you need an nginx service and also an nginx-openresty service. You have to create a new package that changes all the paths to have, for example, an `-openresty` suffix.

Or suppose that you want to run two different instances of mysql: 5.2 and 5.5. The same thing applies, plus you have to also make sure the two mysqlclient libraries do not collide.

This is not impossible but it *is* very inconvenient. If you want to install two whole stacks of software like GNOME 3.10 and GNOME 3.12, you can imagine the amount of work.

From an administrator's point of view: you can use containers. The typical solution nowadays is to create a container per service, especially when different versions are needed. That somewhat solves the problem, but at a different level and with other drawbacks. For example, needing orchestration tools, setting up a shared cache of packages, and new machines to monitor rather than simple services.

From a developer's point of view: you can use virtualenv for python, or jhbuild for gnome, or whatever else. But then how do you mix the two stacks? How do you avoid recompiling the same thing when it could instead be shared? Also you need to set up your development tools to point to the different directories where libraries are installed. Not only that, there's the risk that some of the software incorrectly uses system libraries.

And so on. Nix solves all this at the packaging level and solves it well. A single tool to rule them all.

Being purely functional

Nix makes no assumptions about the global state of the system. This has many advantages, but also some drawbacks of course. The core of a Nix system is the Nix store, usually installed under `/nix/store`, and some tools to manipulate the store. In Nix there is the notion of a *derivation* rather than a package. The difference can be subtle at the beginning, so I will often use the words interchangeably.

Derivations/packages are stored in the Nix store as follows: `/nix/store/«hash-name»`, where the hash uniquely identifies the derivation (this isn't quite true, it's a little more complex), and the name is the name of the derivation.

Let's take a bash derivation as an example: `/nix/store/s4zia7hhqkin1di0f187b79sa2srhv6k-bash-4.2-p45/`. This is a directory in the Nix store which contains `bin/bash`.

What that means is that there's no `/bin/bash`, there's only that self-contained build output in the store. The same goes for coreutils and everything else. To make them convenient to use from the shell, Nix will arrange for binaries to appear in your `PATH` as appropriate.

What we have is basically a store of all packages (with different versions occupying different locations), and everything in the Nix store is immutable.

In fact, there's no ldconfig cache either. So where does bash find libc?

```
$ ldd `which bash`
libc.so.6 => /nix/store/94n64qy99ja0vgbkf675nyk39g9b978n-glibc-2.19/lib/libc.so.6 (0
```

It turns out that when bash was built, it was built against that specific version of glibc in the Nix store, and at runtime it will require exactly that glibc version.

Don't be confused by the version in the derivation name: it's only a name for us humans. You may end up having two derivations with the same name but different hashes: it's the hash that really matters.

What does all this mean? It means that you could run mysql 5.2 with glibc–2.18, and mysql 5.5 with glibc–2.19. You could use your python module with python 2.7 compiled with gcc 4.6 and the same python module with python 3 compiled with gcc 4.8, all in the same system.

In other words: no dependency hell, not even a dependency resolution algorithm. Straight dependencies from derivations to other derivations.

From an administrator's point of view: if you want an old PHP version for one application, but want to upgrade the rest of the system, that's not painful any more.

From a developer's point of view: if you want to develop webkit with llvm 3.4 and 3.3, that's not painful any more.

Mutable vs. immutable

When upgrading a library, most package managers replace it in–place. All new applications run afterwards with the new library without being recompiled. After all, they all refer dynamically to `libc6.so`.

Since Nix derivations are immutable, upgrading a library like glibc means recompiling all applications, because the glibc path to the Nix store has been hardcoded.

So how do we deal with security updates? In Nix we have some tricks (still pure) to solve this problem, but that's another story.

Another problem is that unless software has in mind a pure functional model, or can be adapted to it, it can be hard to compose applications at runtime.

Let's take Firefox for example. On most systems, you install flash, and it starts working in Firefox because Firefox looks in a global path for plugins.

In Nix, there's no such global path for plugins. Firefox therefore must know explicitly about the path to flash. The way we handle this problem is to wrap the Firefox binary so that we can setup the necessary environment to make it find flash in the nix store. That will produce a new Firefox derivation: be aware that it takes a few seconds, and it makes composition harder at runtime.

There are no upgrade/downgrade scripts for your data. It doesn't make sense with this approach, because there's no real derivation to be upgraded. With Nix you switch to using other software with its own stack of dependencies, but there's no formal notion of upgrade or downgrade when doing so.

If there is a data format change, then migrating to the new data format remains your own responsibility.

Conclusion

Nix lets you compose software at build time with maximum flexibility, and with builds being as reproducible as possible. Not only that, due to its nature deploying systems in the cloud is so easy, consistent, and reliable that in the Nix world all existing self–containment and orchestration tools are deprecated by NixOps.

It however *currently* falls short when working with dynamic composition at runtime or replacing low level libraries, due to the need to rebuild dependencies.

That may sound scary, however after running NixOS on both a server and a laptop desktop, I'm very satisfied so far. Some of the architectural problems just need some man–power, other design problems still need to be solved as a community.

Considering Nixpkgs (github link) is a completely new repository of all the existing software, with a completely fresh concept, and with few core developers but overall year–over–year increasing contributions, the current state is more than acceptable and beyond the experimental stage. In other words, it's worth your investment.

Next pill...

...we will install Nix on top of your current system (I assume GNU/Linux, but we also have OSX users) and start inspecting the installed software.

Install on Your Running System

Welcome to the second Nix pill. In the first pill we briefly described Nix.

Now we'll install Nix on our running system and understand what changed in our system after the installation. *If you're using NixOS, Nix is already installed; you can skip to the next pill.*

For installation instructions, please refer to the Nix Reference Manual on Installing Nix.

Installation

These articles are not a tutorial on *using* Nix. Instead, we're going to walk through the Nix system to understand the fundamentals.

The first thing to note: derivations in the Nix store refer to other derivations which are themselves in the Nix store. They don't use `libc` from our system or anywhere else. It's a self–contained store of all the software we need to bootstrap up to any particular package.

Note: In a multi–user installation, such as the one used in NixOS, the store is owned by root and multiple users can install and build software through a Nix daemon. You can read more about multi–user installations here.

The beginnings of the Nix store

Start looking at the output of the install command:

```
copying Nix to /nix/store..........................
```

That's the `/nix/store` we were talking about in the first article. We're copying in the necessary software to bootstrap a Nix system. You can see bash, coreutils, the C compiler toolchain, perl libraries, sqlite and Nix itself with its own tools and libnix.

You may have noticed that `/nix/store` can contain not only directories, but also files, still always in the form «hash–name».

The Nix database

Right after copying the store, the installation process initializes a database:

```
initialising Nix database...
```

Yes, Nix also has a database. It's stored under /nix/var/nix/db. It is a sqlite database that keeps track of the dependencies between derivations.

The schema is very simple: there's a table of valid paths, mapping from an auto increment integer to a store path.

Then there's a dependency relation from path to paths upon which they depend.

You can inspect the database by installing sqlite (nix-env -iA sqlite -f '<nixpkgs>') and then running sqlite3 /nix/var/nix/db/db.sqlite.

Note: If this is the first time you're using Nix after the initial installation, remember you must close and open your terminals first, so that your shell environment will be updated.

Important: Never change /nix/store manually. If you do, then it will no longer be in sync with the sqlite db, unless you *really* know what you are doing.

The first profile

Next in the installation, we encounter the concept of the profile:

```
creating /home/nix/.nix-profile
installing 'nix-2.1.3'
building path(s) `/nix/store/a7p1w3z2h8pl00ywvw6icr3g5l9vm5r7-user-environment'
created 7 symlinks in user environment
```

A profile in Nix is a general and convenient concept for realizing rollbacks. Profiles are used to compose components that are spread among multiple paths under a new unified path. Not only that, but profiles are made up of multiple "generations": they are versioned. Whenever you change a profile, a new generation is created.

Generations can be switched and rolled back atomically, which makes them convenient for managing changes to your system.

Let's take a closer look at our profile:

```
$ ls -l ~/.nix-profile/
bin -> /nix/store/ig31y9gfpp8pf3szdd7d4sf29zr7igbr-nix-2.1.3/bin
[...]
manifest.nix -> /nix/store/q8b5238akq07lj9gfb3qb5ycq4dxxiwm-env-manifest.nix
[...]
share -> /nix/store/ig31y9gfpp8pf3szdd7d4sf29zr7igbr-nix-2.1.3/share
```

That nix-2.1.3 derivation in the Nix store is Nix itself, with binaries and libraries. The process of "installing" the derivation in the profile basically reproduces the hierarchy of the nix-2.1.3 store derivation in the profile by means of symbolic links.

The contents of this profile are special, because only one program has been installed in our profile, therefore e.g. the bin directory points to the only program which has been installed (Nix itself).

But that's only the contents of the latest generation of our profile. In fact, ~/.nix-profile itself is a symbolic link to /nix/var/nix/profiles/default.

In turn, that's a symlink to default-1-link in the same directory. Yes, that means it's the first generation of the default profile.

Finally, default-1-link is a symlink to the nix store "user-environment" derivation that you saw printed during the installation process.

We'll talk about manifest.nix more in the next article.

Nixpkgs expressions

More output from the installer:

```
downloading Nix expressions from `http://releases.nixos.org/nixpkgs/nixpkgs-14.10pre
unpacking channels...
created 2 symlinks in user environment
modifying /home/nix/.profile...
```

Nix expressions are written in the Nix language and used to describe packages and how to build them. Nixpkgs is the repository containing all of the expressions: https://github.com/NixOS/nixpkgs.

The installer downloaded the package descriptions from commit `a1a2851`.

The second profile we discover is the channels profile. `~/.nix-defexpr/channels` points to `/nix/var/nix/profiles/per-user/nix/channels` which points to `channels-1-link` which points to a Nix store directory containing the downloaded Nix expressions.

Channels are a set of packages and expressions available for download. Similar to Debian stable and unstable, there's a stable and unstable channel. In this installation, we're tracking `nixpkgs-unstable`.

Don't worry about Nix expressions yet, we'll get to them later.

Finally, for your convenience, the installer modified `~/.profile` to automatically enter the Nix environment. What `~/.nix-profile/etc/profile.d/nix.sh` really does is simply to add `~/.nix-profile/bin` to `PATH` and `~/.nix-defexpr/channels/nixpkgs` to `NIX_PATH`. We'll discuss `NIX_PATH` later.

Read `nix.sh`, it's short.

FAQ: Can I change /nix to something else?

You can, but there's a good reason to keep using `/nix` instead of a different directory. All the derivations depend on other derivations by using absolute paths. We saw in the first article that bash referenced a `glibc` under a specific absolute path in `/nix/store`.

You can see for yourself, don't worry if you see multiple bash derivations:

```
$ ldd /nix/store/*bash*/bin/bash
[...]
```

Keeping the store in `/nix` means we can grab the binary cache from nixos.org (just like you grab packages from debian mirrors) otherwise:

- `glibc` would be installed under `/foo/store`
- Thus bash would need to point to `glibc` under `/foo/store`, instead of under `/nix/store`
- So the binary cache can't help, because we need a *different* bash, and so we'd have to recompile everything ourselves.

After all `/nix` is a sensible place for the store.

Conclusion

We've installed Nix on our system, fully isolated and owned by the `nix` user as we're still coming to terms with this new system.

We learned some new concepts like profiles and channels. In particular, with profiles we're able to manage multiple generations of a composition of packages, while with channels we're able to download binaries from `nixos.org`.

The installation put everything under `/nix`, and some symlinks in the Nix user home. That's because every user is able to install and use software in her own environment.

I hope I left nothing uncovered so that you think there's some kind of magic going on behind the scenes. It's all about putting components in the store and symlinking these components together.

Next pill...

...we will enter the Nix environment and learn how to interact with the store.

Enter the Environment

Welcome to the third Nix pill. In the second pill we installed Nix on our running system. Now we can finally play with it a little, these things also apply to NixOS users.

Enter the environment

If you're using NixOS, you can skip to the next step.

In the previous article we created a Nix user, so let's start by switching to it with `su - nix`. If your `~/.profile` got evaluated, then you should now be able to run commands like `nix-env` and `nix-store`.

If that's not the case:

```
$ source ~/.nix-profile/etc/profile.d/nix.sh
```

To remind you, `~/.nix-profile/etc` points to the `nix-2.1.3` derivation. At this point, we are in our Nix user profile.

Install something

Finally something practical! Installation into the Nix environment is an interesting process. Let's install `hello`, a simple CLI tool which prints `Hello world` and is mainly used to test compilers and package installations.

Back to the installation:

```
$ nix-env -i hello
installing 'hello-2.10'
[...]
building '/nix/store/0vqw0ssmh6y5zj48yg34gc6macr883xk-user-environment.drv'...
created 36 symlinks in user environment
```

Now you can run `hello`. Things to notice:

- We installed software as a user, and only for the Nix user.
- It created a new user environment. That's a new generation of our Nix user profile.
- The nix-env tool manages environments, profiles and their generations.
- We installed `hello` by derivation name minus the version. I repeat: we specified the **derivation name** (minus the version) to install it.

We can list generations without walking through the `/nix` hierarchy:

```
$ nix-env --list-generations
   1   2014-07-24 09:23:30
   2   2014-07-25 08:45:01   (current)
```

Listing installed derivations:

```
$ nix-env -q
nix-2.1.3
hello-2.10
```

So, where did `hello` really get installed? `which hello` is `~/.nix-profile/bin/hello` which points to the store. We can also list the derivation paths with `nix-env -q --out-path`. So that's what those derivation paths are called: the **output** of a build.

Path merging

At this point you probably want to run `man` to get some documentation. Even if you already have man system–wide outside of the Nix environment, you can install and use it within Nix with `nix-env -i man-db`. As usual, a new generation will be created, and `~/.nix-profile` will point to it.

Let's inspect the profile a bit:

```
$ ls -l ~/.nix-profile/
dr-xr-xr-x 2 nix nix 4096 Jan  1  1970 bin
lrwxrwxrwx 1 nix nix   55 Jan  1  1970 etc -> /nix/store/ig31y9gfpp8pf3szdd7d4sf29zr
[...]
```

Now that's interesting. When only `nix-2.1.3` was installed, `bin` was a symlink to `nix-2.1.3`. Now that we've actually installed some things (`man`, `hello`), it's a real directory, not a symlink.

```
$ ls -l ~/.nix-profile/bin/
[...]
man -> /nix/store/83cn9ing5sc6644h50dqzzfxcs07r2jn-man-1.6g/bin/man
[...]
nix-env -> /nix/store/ig31y9gfpp8pf3szdd7d4sf29zr7igbr-nix-2.1.3/bin/nix-env
[...]
hello -> /nix/store/58r35bqb4f3lxbnbabq718svq9i2pda3-hello-2.10/bin/hello
[...]
```

Okay, that's clearer now. `nix-env` merged the paths from the installed derivations. `which man` points to the Nix profile, rather than the system `man`, because `~/.nix-profile/bin` is at the head of `$PATH`.

Rolling back and switching generation

The last command installed `man`. We should be at generation 3, unless you changed something in the middle. Let's say we want to rollback to the old generation:

```
$ nix-env --rollback
switching from generation 3 to 2
```

Now `nix-env -q` does not list `man` anymore. ``ls -l `which man` `` should now be your system copy.

Enough with the rollback, let's go back to the most recent generation:

```
$ nix-env -G 3
switching from generation 2 to 3
```

I invite you to read the manpage of `nix-env`. `nix-env` requires an operation to perform, then there are common options for all operations, as well as options specific to each operation.

You can of course also uninstall and upgrade packages.

Querying the store

So far we learned how to query and manipulate the environment. But all of the environment components point to the store.

To query and manipulate the store, there's the `nix-store` command. We can do some interesting things, but we'll only see some queries for now.

To show the direct runtime dependencies of `hello`:

```
$ nix-store -q --references `which hello`
/nix/store/fg4yq8i8wd08xg3fy58l6q73cjy8hjr2-glibc-2.27
/nix/store/58r35bqb4f3lxbnbabq718svq9i2pda3-hello-2.10
```

The argument to `nix-store` can be anything as long as it points to the Nix store. It will follow symlinks.

It may not make sense to you right now, but let's print reverse dependencies of `hello`:

```
$ nix-store -q --referrers `which hello`
/nix/store/58r35bqb4f3lxbnbabq718svq9i2pda3-hello-2.10
/nix/store/fhvy2550cpmjgcjcx5rzz328i0kfv3z3-env-manifest.nix
/nix/store/yzdk0xvr0b8dcwhi2nns6d75k2ha5208-env-manifest.nix
/nix/store/mp987abm20c70pl8p31ljw1r5by4xwfw-user-environment
/nix/store/ppr3qbq7fk2m2pa49i2z3i32cvfhsv7p-user-environment
```

Was it what you expected? It turns out that our environments depend upon `hello`. Yes, that means that the environments are in the store, and since they contain symlinks to `hello`, therefore the environment depends upon `hello`.

Two environments were listed, generation 2 and generation 3, since these are the ones that had `hello` installed in them.

The `manifest.nix` file contains metadata about the environment, such as which derivations are installed. So that `nix-env` can list, upgrade or remove them. And yet again, the current `manifest.nix` can be found at `~/.nix-profile/manifest.nix`.

Closures

The closures of a derivation is a list of all its dependencies, recursively, including absolutely everything necessary to use that derivation.

```
$ nix-store -qR `which man`
[...]
```

Copying all those derivations to the Nix store of another machine makes you able to run `man` out of the box on that other machine. That's the base of deployment using Nix, and you can already foresee the potential when deploying software in the cloud (hint: `nix-copy-closures` and `nix-store --export`).

A nicer view of the closure:

```
$ nix-store -q --tree `which man`
[...]
```

With the above command, you can find out exactly why a *runtime* dependency, be it direct or indirect, exists for a given derivation.

The same applies to environments. As an exercise, run `nix-store -q --tree ~/.nix-profile`, and see that the first children are direct dependencies of the user environment: the installed derivations, and the `manifest.nix`.

Dependency resolution

There isn't anything like `apt` which solves a SAT problem in order to satisfy dependencies with lower and upper bounds on versions. There's no need for this because all the dependencies are static: if a derivation X depends on a derivation Y, then it always depends on it. A version of X which depended on Z would be a different derivation.

Recovering the hard way

```
$ nix-env -e '*'
uninstalling 'hello-2.10'
uninstalling 'nix-2.1.3'
[...]
```

Oops, that uninstalled all derivations from the environment, including Nix. That means we can't even run `nix-env`, what now?

Previously we got `nix-env` from the environment. Environments are a convenience for the user, but Nix is still there in the store!

First, pick one `nix-2.1.3` derivation: `ls /nix/store/*nix-2.1.3`, say `/nix/store/ig31y9gfpp8pf3szdd7d4sf29zr7igbr-nix-2.1.3`.

The first option is to rollback:

```
$ /nix/store/ig31y9gfpp8pf3szdd7d4sf29zr7igbr-nix-2.1.3/bin/nix-env --rollback
```

The second option is to install Nix, thus creating a new generation:

```
$ /nix/store/ig31y9gfpp8pf3szdd7d4sf29zr7igbr-nix-2.1.3/bin/nix-env -i /nix/store/ig
```

Channels

So where are we getting packages from? We said something about this already in the second article. There's a list of channels from which we get packages, although usually we use a single channel. The tool to manage channels is nix-channel.

```
$ nix-channel --list
nixpkgs http://nixos.org/channels/nixpkgs-unstable
```

If you're using NixOS, you may not see any output from the above command (if you're using the default), or you may see a channel whose name begins with "nixos-" instead of "nixpkgs".

That's essentially the contents of `~/.nix-channels`.

Note: `~/.nix-channels` is not a symlink to the nix store!

To update the channel run `nix-channel --update`. That will download the new Nix expressions (descriptions of the packages), create a new generation of the channels profile and unpack it under `~/.nix-defexpr/channels`.

This is quite similar to `apt-get update`. (See this table for a rough mapping between Ubuntu and NixOS package management.)

Conclusion

We learned how to query the user environment and to manipulate it by installing and uninstalling software. Upgrading software is also straightforward, as you can read in the manual (`nix-env -u` will upgrade all packages in the environment).

Every time we change the environment, a new generation is created. Switching between generations is easy and immediate.

Then we learned how to query the store. We inspected the dependencies and reverse dependencies of store paths.

We saw how symlinks are used to compose paths from the Nix store, a useful trick.

A quick analogy with programming languages: you have the heap with all the objects, that corresponds to the Nix store. You have objects that point to other objects, those correspond to derivations. This is a suggestive metaphor, but will it be the right path?

Next pill

...we will learn the basics of the Nix language. The Nix language is used to describe how to build derivations, and it's the basis for everything else, including NixOS. Therefore it's very important to understand both the syntax and the semantics of the language.

The Basics of the Language

Welcome to the fourth Nix pill. In the previous article we learned about Nix environments. We installed software as a user, managed their profile, switched between generations, and queried the Nix store. Those are the very basics of system administration using Nix.

The Nix language is used to write expressions that produce derivations. The nix-build tool is used to build derivations from an expression. Even as a system administrator that wants to customize the installation, it's necessary to master Nix. Using Nix for your jobs means you get the features we saw in the previous articles for free.

The syntax of Nix is quite unfamiliar, so looking at existing examples may lead you to think that there's a lot of magic happening. In reality, it's mostly about writing utility functions to make things convenient.

On the other hand, the same syntax is great for describing packages, so learning the language itself will pay off when writing package expressions.

Important: In Nix, everything is an expression, there are no statements. This is common in functional languages.

Important: Values in Nix are immutable.

Value types

Nix 2.0 contains a command named `nix repl` which is a simple command line tool for playing with the Nix language. In fact, Nix is a pure, lazy, functional language, not only a set of tools to manage derivations. The `nix repl` syntax is slightly different to Nix syntax when it comes to assigning variables, but it shouldn't be confusing so long as you bear it in mind. I prefer to start with `nix repl` before cluttering your mind with more complex expressions.

Launch `nix repl`. First of all, Nix supports basic arithmetic operations: `+`, `-`, `*` and `/`. (To exit `nix repl`, use the command `:q`. Help is available through the `:?` command.)

```
nix-repl> 1+3
4

nix-repl> 7-4
3

nix-repl> 3*2
6
```

Attempting to perform division in Nix can lead to some surprises.

```
nix-repl> 6/3
/home/nix/6/3
```

What happened? Recall that Nix is not a general purpose language, it's a domain-specific language for writing packages. Integer division isn't actually that useful when writing package expressions. Nix parsed `6/3` as a relative path to the current directory. To get Nix to perform division instead, leave a space after the `/`. Alternatively, you can use `builtins.div`.

```
nix-repl> 6/ 3
2

nix-repl> builtins.div 6 3
2
```

Other operators are `||`, `&&` and `!` for booleans, and relational operators such as `!=`, `==`, `<`, `>`, `<=`, `>=`. In Nix, `<`, `>`, `<=` and `>=` are not much used. There are also other operators we will see in the course of this series.

Nix has integer, floating point, string, path, boolean and null simple types. Then there are also lists, sets and functions. These types are enough to build an operating system.

Nix is strongly typed, but it's not statically typed. That is, you cannot mix strings and integers, you must first do the conversion.

As demonstrated above, expressions will be parsed as paths as long as there's a slash not followed by a space. Therefore to specify the current directory, use `./`. In addition, Nix also parses urls specially.

Not all urls or paths can be parsed this way. If a syntax error occurs, it's still possible to fallback to plain strings. Literal urls and paths are convenient for additional safety.

Identifier

There's not much to say here, except that dash (`-`) is allowed in identifiers. That's convenient since many packages use dash in their names. In fact:

```
nix-repl> a-b
error: undefined variable `a-b' at (string):1:1
nix-repl> a - b
error: undefined variable `a' at (string):1:1
```

As you can see, `a-b` is parsed as identifier, not as a subtraction.

Strings

It's important to understand the syntax for strings. When learning to read Nix expressions, you may find dollars (`$`) ambiguous, but they are very important . Strings are enclosed by double quotes (`"`), or two single quotes (`''`).

```
nix-repl> "foo"
"foo"
nix-repl> ''foo''
"foo"
```

In other languages like Python you can also use single quotes for strings (e.g. `'foo'`), but not in Nix.

It's possible to interpolate whole Nix expressions inside strings with the `${...}` syntax and only that syntax, not `$foo` or `{$foo}` or anything else.

```
nix-repl> foo = "strval"
nix-repl> "$foo"
"$foo"
nix-repl> "${foo}"
"strval"
nix-repl> "${2+3}"
error: cannot coerce an integer to a string, at (string):1:2
```

Note: ignore the `foo = "strval"` assignment, special syntax in `nix repl`.

As said previously, you cannot mix integers and strings. You need to explicitly include conversions. We'll see this later: function calls are another story.

Using the syntax with two single quotes is useful for writing double quotes inside strings without needing to escape them:

```
nix-repl> ''test " test''
"test \" test"
nix-repl> ''${foo}''
"strval"
```

Escaping `${...}` within double quoted strings is done with the backslash. Within two single quotes, it's done with `''`:

```
nix-repl> "\${foo}"
"${foo}"
nix-repl> ''test ''${foo} test''
"test ${foo} test"
```

Lists

Lists are a sequence of expressions delimited by space (*not* comma):

```
nix-repl> [ 2 "foo" true (2+3) ]
[ 2 "foo" true 5 ]
```

Lists, like everything else in Nix, are immutable. Adding or removing elements from a list is possible, but will return a new list.

Attribute sets

An attribute set is an association between string keys and Nix values. Keys can only be strings. When writing attribute sets you can also use unquoted identifiers as keys.

```
nix-repl> s = { foo = "bar"; a-b = "baz"; "123" = "num"; }
nix-repl> s
{ "123" = "num"; a-b = "baz"; foo = "bar"; }
```

For those reading Nix expressions from nixpkgs: do not confuse attribute sets with argument sets used in functions.

To access elements in the attribute set:

```
nix-repl> s.a-b
"baz"
nix-repl> s."123"
"num"
```

Yes, you can use strings to address keys which aren't valid identifiers.

Inside an attribute set you cannot normally refer to elements of the same attribute set:

```
nix-repl> { a = 3; b = a+4; }
error: undefined variable `a' at (string):1:10
```

To do so, use recursive attribute sets:

```
nix-repl> rec { a = 3; b = a+4; }
{ a = 3; b = 7; }
```

This is very convenient when defining packages, which tend to be recursive attribute sets.

If expressions

These are expressions, not statements.

```
nix-repl> a = 3
nix-repl> b = 4
nix-repl> if a > b then "yes" else "no"
"no"
```

You can't have only the `then` branch, you must specify also the `else` branch, because an expression must have a value in all cases.

Let expressions

This kind of expression is used to define local variables for inner expressions.

```
nix-repl> let a = "foo"; in a
"foo"
```

The syntax is: first assign variables, then `in`, then an expression which can use the defined variables. The value of the whole `let` expression will be the value of the expression after the `in`.

```
nix-repl> let a = 3; b = 4; in a + b
7
```

Let's write two `let` expressions, one inside the other:

```
nix-repl> let a = 3; in let b = 4; in a + b
7
```

With `let` you cannot assign twice to the same variable. However, you can shadow outer variables:

```
nix-repl> let a = 3; a = 8; in a
error: attribute `a' at (string):1:12 already defined at (string):1:5
nix-repl> let a = 3; in let a = 8; in a
8
```

You cannot refer to variables in a `let` expression outside of it:

```
nix-repl> let a = (let c = 3; in c); in c
error: undefined variable `c' at (string):1:31
```

You can refer to variables in the `let` expression when assigning variables, like with recursive attribute sets:

```
nix-repl> let a = 4; b = a + 5; in b
9
```

So beware when you want to refer to a variable from the outer scope, but it's also defined in the current let expression. The same applies to recursive attribute sets.

With expression

This kind of expression is something you rarely see in other languages. You can think of it like a more granular version of `using` from C++, or `from module import *` from Python. You decide per–expression when to include symbols into the scope.

```
nix-repl> longName = { a = 3; b = 4; }
nix-repl> longName.a + longName.b
7
nix-repl> with longName; a + b
7
```

That's it, it takes an attribute set and includes symbols from it in the scope of the inner expression. Of course, only valid identifiers from the keys of the set will be included. If a symbol exists in the outer scope and would also be introduced by the `with`, it will *not* be shadowed. You can however still refer to the attribute set:

```
nix-repl> let a = 10; in with longName; a + b
14
nix-repl> let a = 10; in with longName; longName.a + b
7
```

Laziness

Nix evaluates expressions only when needed. This is a great feature when working with packages.

```
nix-repl> let a = builtins.div 4 0; b = 6; in b
6
```

Since `a` is not needed, there's no error about division by zero, because the expression is not in need to be evaluated. That's why we can have all the packages defined on demand, yet have access to specific packages very quickly.

Next pill

...we will talk about functions and imports. In this pill I've tried to avoid function calls as much as possible, otherwise the post would have been too long.

Functions and Imports

Welcome to the fifth Nix pill. In the previous fourth pill we touched the Nix language for a moment. We introduced basic types and values of the Nix language, and basic expressions such as `if`, `with` and `let`. I invite you to re–read about these expressions and play with them in the repl.

Functions help to build reusable components in a big repository like nixpkgs. The Nix manual has a great explanation of functions. Let's go: pill on one hand, Nix manual on the other hand.

I remind you how to enter the Nix environment: `source ~/.nix-profile/etc/profile.d/nix.sh`

Nameless and single parameter

Functions are anonymous (lambdas), and only have a single parameter. The syntax is extremely simple. Type the parameter name, then "`:`", then the body of the function.

```
nix-repl> x: x*2
«lambda»
```

So here we defined a function that takes a parameter `x`, and returns `x*2`. The problem is that we cannot use it in any way, because it's unnamed... joke!

We can store functions in variables.

```
nix-repl> double = x: x*2
nix-repl> double
«lambda»
nix-repl> double 3
6
```

As usual, please ignore the special syntax for assignments inside `nix repl`. So, we defined a function `x: x*2` that takes one parameter `x`, and returns `x*2`. This function is then assigned to the variable `double`. Finally we did our first function call: `double 3`.

Big note: it's not like many other programming languages where you write `double(3)`. It really is `double 3`.

In summary: to call a function, name the variable, then space, then the argument. Nothing else to say, it's as easy as that.

More than one parameter

How do we create a function that accepts more than one parameter? For people not used to functional programming, this may take a while to grasp. Let's do it step by step.

```
nix-repl> mul = a: (b: a*b)
nix-repl> mul
«lambda»
nix-repl> mul 3
«lambda»
nix-repl> (mul 3) 4
12
```

We defined a function that takes the parameter `a`, the body returns another function. This other function takes a parameter `b` and returns `a*b`. Therefore, calling `mul 3` returns this kind of function: `b: 3*b`. In turn, we call the returned function with `4`, and get the expected result.

You don't have to use parentheses at all, Nix has sane priorities when parsing the code:

```
nix-repl> mul = a: b: a*b
nix-repl> mul
«lambda»
nix-repl> mul 3
«lambda»
nix-repl> mul 3 4
12
nix-repl> mul (6+7) (8+9)
221
```

Much more readable, you don't even notice that functions only receive one argument. Since the argument is separated by a space, to pass more complex expressions you need parentheses. In other common languages you would write `mul(6+7, 8+9)`.

Given that functions have only one parameter, it is straightforward to use **partial application**:

```
nix-repl> foo = mul 3
nix-repl> foo 4
12
nix-repl> foo 5
15
```

We stored the function returned by `mul 3` into a variable foo, then reused it.

Argument set

Now this is a very cool feature of Nix. It is possible to pattern match over a set in the parameter. We write an alternative version of `mul = a: b: a*b` first by using a set as argument, then using pattern matching.

```
nix-repl> mul = s: s.a*s.b
nix-repl> mul { a = 3; b = 4; }
12
nix-repl> mul = { a, b }: a*b
nix-repl> mul { a = 3; b = 4; }
12
```

In the first case we defined a function that accepts a single parameter. We then access attributes `a` and `b` from the given set. Note how the parentheses–less syntax for function calls is very elegant in this case, instead of doing `mul({ a=3; b=4; })` in other languages.

In the second case we defined an argument set. It's like defining a set, except without values. We require that the passed set contains the keys `a` and `b`. Then we can use those `a` and `b` in the function body directly.

```
nix-repl> mul = { a, b }: a*b
nix-repl> mul { a = 3; b = 4; c = 6; }
error: anonymous function at (string):1:2 called with unexpected argument `c', at (s
nix-repl> mul { a = 3; }
error: anonymous function at (string):1:2 called without required argument `b', at (
```

Only a set with exactly the attributes required by the function is accepted, nothing more, nothing less.

Default and variadic attributes

It is possible to specify **default values** of attributes in the argument set:

```
nix-repl> mul = { a, b ? 2 }: a*b
nix-repl> mul { a = 3; }
6
nix-repl> mul { a = 3; b = 4; }
12
```

Also you can allow passing more attributes (**variadic**) than the expected ones:

```
nix-repl> mul = { a, b, ... }: a*b
nix-repl> mul { a = 3; b = 4; c = 2; }
```

However, in the function body you cannot access the "c" attribute. The solution is to give a name to the given set with the **@–pattern**:

```
nix-repl> mul = s@{ a, b, ... }: a*b*s.c
nix-repl> mul { a = 3; b = 4; c = 2; }
24
```

That's it, you give a name to the whole parameter with name@ before the set pattern.

Advantages of using argument sets:

- Named unordered arguments: you don't have to remember the order of the arguments.
- You can pass sets, that adds a whole new layer of flexibility and convenience.

Disadvantages:

- Partial application does not work with argument sets. You have to specify the whole attribute set, not part of it.

You may find similarities with Python **kwargs.

Imports

The `import` function is built–in and provides a way to parse a `.nix` file. The natural approach is to define each component in a `.nix` file, then compose by importing these files.

Let's start with the bare metal.

`a.nix`:

```
3
```

`b.nix`:

```
4
```

`mul.nix`:

```
a: b: a*b
```

```
nix-repl> a = import ./a.nix
nix-repl> b = import ./b.nix
nix-repl> mul = import ./mul.nix
nix-repl> mul a b
12
```

Yes it's really that simple. You import a file, and it gets parsed as an expression. Note that the scope of the imported file does not inherit the scope of the importer.

`test.nix`:

```
x
```

```
nix-repl> let x = 5; in import ./test.nix
error: undefined variable `x' at /home/lethal/test.nix:1:1
```

So how do we pass information to the module? Use functions, like we did with `mul.nix`. A more complex example:

`test.nix`:

```
{ a, b ? 3, trueMsg ? "yes", falseMsg ? "no" }:
if a > b
  then builtins.trace trueMsg true
  else builtins.trace falseMsg false
```

```
nix-repl> import ./test.nix { a = 5; trueMsg = "ok"; }
trace: ok
true
```

Explaining:

- In `test.nix` we return a function. It accepts a set, with default attributes `b`, `trueMsg` and `falseMsg`.
- `builtins.trace` is a built–in function that takes two arguments. The first is the message to display, the second is the value to return. It's usually used for debugging purposes.
- Then we import `test.nix`, and call the function with that set.

So when is the message shown? Only when it needs to be evaluated.

Next pill

...we will finally write our first derivation.

Our First Derivation

Welcome to the sixth Nix pill. In the previous fifth pill we introduced functions and imports. Functions and imports are very simple concepts that allow for building complex abstractions and composition of modules to build a flexible Nix system.

In this post we finally arrived to writing a derivation. Derivations are the building blocks of a Nix system, from a file system view point. The Nix language is used to describe such derivations.

I remind you how to enter the Nix environment: `source ~/.nix-profile/etc/profile.d/nix.sh`

The derivation function

The derivation built–in function is used to create derivations. I invite you to read the link in the Nix manual about the derivation built–in. A derivation from a Nix language view point is simply a set, with some attributes. Therefore you can pass the derivation around with variables like anything else.

That's where the real power comes in.

The `derivation` function receives a set as its first argument. This set requires at least the following three attributes:

- name: the name of the derivation. In the nix store the format is hash−name, that's the name.
- system: is the name of the system in which the derivation can be built. For example, x86$_{64}$−linux.
- builder: is the binary program that builds the derivation.

First of all, what's the name of our system as seen by nix?

```
nix-repl> builtins.currentSystem
"x86_64-linux"
```

Let's try to fake the name of the system:

```
nix-repl> d = derivation { name = "myname"; builder = "mybuilder"; system = "mysyste
nix-repl> d
«derivation /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv»
```

Oh oh, what's that? Did it build the derivation? No it didn't, but it **did create the .drv file**. `nix repl` does not build derivations unless you tell it to do so.

Digression about .drv files

What's that `.drv` file? It is the specification of how to build the derivation, without all the Nix language fuzz.

Before continuing, some analogies with the C language:

- `.nix` files are like `.c` files.
- `.drv` files are intermediate files like `.o` files. The `.drv` describes how to build a derivation; it's the bare minimum information.
- out paths are then the product of the build.

Both drv paths and out paths are stored in the nix store as you can see.

What's in that `.drv` file? You can read it, but it's better to pretty print it:

Note: If your version of nix doesn't have `nix derivation show`, use `nix show-derivation` instead.

```
$ nix derivation show /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv
{
  "/nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv": {
    "outputs": {
      "out": {
        "path": "/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname"
      }
    },
    "inputSrcs": [],
    "inputDrvs": {},
    "platform": "mysystem",
    "builder": "mybuilder",
    "args": [],
    "env": {
      "builder": "mybuilder",
      "name": "myname",
      "out": "/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname",
      "system": "mysystem"
```

```
    }
  }
}
```

Ok, we can see there's an out path, but it does not exist yet. We never told Nix to build it, but we know beforehand where the build output will be. Why?

Think, if Nix ever built the derivation just because we accessed it in Nix, we would have to wait a long time if it was, say, Firefox. That's why Nix let us know the path beforehand and kept evaluating the Nix expressions, but it's still empty because no build was ever made.

Important: the hash of the out path is based solely on the input derivations in the current version of Nix, not on the contents of the build product. It's possible however to have content−addressable derivations for e.g. tarballs as we'll see later on.

Many things are empty in that `.drv`, however I'll write a summary of the .drv format for you:

1. The output paths (there can be multiple ones). By default nix creates one out path called "out".
2. The list of input derivations. It's empty because we are not referring to any other derivation. Otherwise, there would be a list of other .drv files.
3. The system and the builder executable (yes, it's a fake one).
4. Then a list of environment variables passed to the builder.

That's it, the minimum necessary information to build our derivation.

Important note: the environment variables passed to the builder are just those you see in the .drv plus some other Nix related configuration (number of cores, temp dir, ...). The builder will not inherit any variable from your running shell, otherwise builds would suffer from non−determinism.

Back to our fake derivation.

Let's build our really fake derivation:

```
nix-repl> d = derivation { name = "myname"; builder = "mybuilder"; system = "mysyste
nix-repl> :b d
[...]
these derivations will be built:
  /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv
building path(s) `/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname'
error: a `mysystem' is required to build `/nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754
```

The `:b` is a `nix repl` specific command to build a derivation. You can see more commands with `:?` . So in the output you can see that it takes the `.drv` as information on how to build the derivation. Then it says it's trying to produce our out path. Finally the error we were waiting for: that derivation can't be built on our system.

We're doing the build inside `nix repl`, but what if we don't want to use `nix repl`? You can **realise** a `.drv` with:

```
$ nix-store -r /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv
```

You will get the same output as before.

Let's fix the system attribute:

```
nix-repl> d = derivation { name = "myname"; builder = "mybuilder"; system = builtins
nix-repl> :b d
[...]
build error: invalid file name `mybuilder'
```

A step forward: of course, that `mybuilder` executable does not really exist. Stop for a moment.

What's in a derivation set

It is useful to start by inspecting the return value from the derivation function. In this case, the returned value is a plain set:

```
nix-repl> d = derivation { name = "myname"; builder = "mybuilder"; system = "mysyste
nix-repl> builtins.isAttrs d
true
nix-repl> builtins.attrNames d
[ "all" "builder" "drvAttrs" "drvPath" "name" "out" "outPath" "outputName" "system"
```

You can guess what `builtins.isAttrs` does; it returns true if the argument is a set. While `builtins.attrNames` returns a list of keys of the given set. Some kind of reflection, you might say.

Start from drvAttrs:

```
nix-repl> d.drvAttrs
{ builder = "mybuilder"; name = "myname"; system = "mysystem"; }
```

That's basically the input we gave to the derivation function. Also the `d.name`, `d.system` and `d.builder` attributes are exactly the ones we gave as input.

```
nix-repl> (d == d.out)
true
```

So out is just the derivation itself, it seems weird but the reason is that we only have one output from the derivation. That's also the reason why `d.all` is a singleton. We'll see multiple outputs later.

The `d.drvPath` is the path of the `.drv` file: `/nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv`.

Something interesting is the `type` attribute. It's `"derivation"`. Nix does add a little of magic to sets with type derivation, but not that much. To help you understand, you can create yourself a set with that type, it's a simple set:

```
nix-repl> { type = "derivation"; }
«derivation ???»
```

Of course it has no other information, so Nix doesn't know what to say :–) But you get it, the `type = "derivation"` is just a convention for Nix and for us to understand the set is a derivation.

When writing packages, we are interested in the outputs. The other metadata is needed for Nix to know how to create the drv path and the out path.

The `outPath` attribute is the build path in the nix store: `/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname`.

Referring to other derivations

Just like dependencies in other package managers, how do we refer to other packages? How do we refer to other derivations in terms of files on the disk? We use the `outPath`. The `outPath` describes the location of the files of that derivation. To make it more convenient, Nix is able to do a conversion from a derivation set to a string.

```
nix-repl> d.outPath
"/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname"
nix-repl> builtins.toString d
"/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname"
```

Nix does the "set to string conversion" as long as there is the `outPath` attribute (much like a toString method in other languages):

```
nix-repl> builtins.toString { outPath = "foo"; }
"foo"
nix-repl> builtins.toString { a = "b"; }
error: cannot coerce a set to a string, at (string):1:1
```

Say we want to use binaries from coreutils (ignore the nixpkgs etc.):

```
nix-repl> :l <nixpkgs>
Added 3950 variables.
nix-repl> coreutils
«derivation /nix/store/1zcs1y4n27lqs0gw4v038i303pb89rw6-coreutils-8.21.drv»
nix-repl> builtins.toString coreutils
"/nix/store/8w4cbiy7wqvaqsnsnb3zvabq1cp2zhyz-coreutils-8.21"
```

Apart from the nixpkgs stuff, just think we added to the scope a series of variables. One of them is coreutils. It is the derivation of the coreutils package you all know of from other Linux distributions. It contains basic binaries for GNU/Linux systems (you may have multiple derivations of coreutils in the nix store, no worries):

```
$ ls /nix/store/*coreutils*/bin
[...]
```

I remind you, inside strings it's possible to interpolate Nix expressions with `${...}`:

```
nix-repl> "${d}"
"/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname"
nix-repl> "${coreutils}"
"/nix/store/8w4cbiy7wqvaqsnsnb3zvabq1cp2zhyz-coreutils-8.21"
```

That's very convenient, because then we could refer to e.g. the bin/true binary like this:

```
nix-repl> "${coreutils}/bin/true"
"/nix/store/8w4cbiy7wqvaqsnsnb3zvabq1cp2zhyz-coreutils-8.21/bin/true"
```

An almost working derivation

In the previous attempt we used a fake builder, `mybuilder` which obviously does not exist. But we can use for example bin/true, which always exits with 0 (success).

```
nix-repl> :l <nixpkgs>
nix-repl> d = derivation { name = "myname"; builder = "${coreutils}/bin/true"; syster
nix-repl> :b d
[...]
builder for `/nix/store/qyfrcd53wmc0v22ymhhd5r6sz5xmdc8a-myname.drv' failed to produ
```

Another step forward, it executed the builder (bin/true), but the builder did not create the out path of course, it just exited with 0.

Obvious note: every time we change the derivation, a new hash is created.

Let's examine the new `.drv` now that we referred to another derivation:

```
$ nix derivation show /nix/store/qyfrcd53wmc0v22ymhhd5r6sz5xmdc8a-myname.drv
{
  "/nix/store/qyfrcd53wmc0v22ymhhd5r6sz5xmdc8a-myname.drv": {
    "outputs": {
      "out": {
        "path": "/nix/store/ly2k1vswbfmswr33hw0kf0ccilrpisnk-myname"
      }
    },
    "inputSrcs": [],
    "inputDrvs": {
      "/nix/store/hixdnzz2wp75x1jy65cysq06yl74vx7q-coreutils-8.29.drv": [
        "out"
      ]
    },
    "platform": "x86_64-linux",
    "builder": "/nix/store/qrxs7sabhqcr3j9ai0j0cp58zfnny0jz-coreutils-8.29/bin/true"
    "args": [],
    "env": {
      "builder": "/nix/store/qrxs7sabhqcr3j9ai0j0cp58zfnny0jz-coreutils-8.29/bin/tru
      "name": "myname",
      "out": "/nix/store/ly2k1vswbfmswr33hw0kf0ccilrpisnk-myname",
      "system": "x86_64-linux"
    }
  }
}
```

Aha! Nix added a dependency to our myname.drv, it's the coreutils.drv. Before doing our build, Nix should build the coreutils.drv. But since coreutils is already in our nix store, no build is needed, it's already there with out path `/nix/store/qrxs7sab-hqcr3j9ai0j0cp58zfnny0jz-coreutils-8.29`.

When is the derivation built

Nix does not build derivations **during evaluation** of Nix expressions. In fact, that's why we have to do ":b drv" in `nix repl`, or use nix−store −r in the first place.

An important separation is made in Nix:

- **Instantiate/Evaluation time**: the Nix expression is parsed, interpreted and finally returns a derivation set. During evaluation, you can refer to other derivations because Nix will create .drv files and we will know out paths beforehand. This is achieved with nix−instantiate.
- **Realise/Build time**: the .drv from the derivation set is built, first building .drv inputs (build dependencies). This is achieved with nix−store −r.

Think of it as of compile time and link time like with C/C++ projects. You first compile all source files to object files. Then link object files in a single executable.

In Nix, first the Nix expression (usually in a .nix file) is compiled to .drv, then each .drv is built and the product is installed in the relative out paths.

Conclusion

Is it that complicated to create a package for Nix? No, it's not.

We're walking through the fundamentals of Nix derivations, to understand how they work, how they are represented. Packaging in Nix is certainly easier than that, but we're not there yet in this post. More Nix pills are needed.

With the derivation function we provide a set of information on how to build a package, and we get back the information about where the package was built. Nix converts a set to a string when there's an `outPath`; that's very convenient. With that, it's easy to refer to other derivations.

When Nix builds a derivation, it first creates a .drv file from a derivation expression, and uses it to build the output. It does so recursively for all the dependencies (inputs). It "executes" the .drv files like a machine. Not much magic after all.

Next pill

...we will finally write our first **working** derivation. Yes, this post is about "our first derivation", but I never said it was a working one ;)

Working Derivation

Introduction

Welcome to the seventh nix pill. In the previous sixth pill we introduced the notion of derivation in the Nix language — how to define a raw derivation and how to (try to) build it.

In this post we continue along the path, by creating a derivation that actually builds something. Then, we try to package a real program: we compile a simple C file and create a derivation out of it, given a blessed toolchain.

I remind you how to enter the Nix environment: `source ~/.nix-profile/etc/profile.d/nix.sh`

Using a script as a builder

What's the easiest way to run a sequence of commands for building something? A bash script. We write a custom bash script, and we want it to be our builder. Given a `builder.sh`, we want the derivation to run `bash builder.sh`.

We don't use hash bangs in `builder.sh`, because at the time we are writing it we do not know the path to bash in the nix store. Yes, even bash is in the nix store, everything is there.

We don't even use /usr/bin/env, because then we lose the cool stateless property of Nix. Not to mention that `PATH` gets cleared when building, so it wouldn't find bash anyway.

In summary, we want the builder to be bash, and pass it an argument, `builder.sh`. Turns out the `derivation` function accepts an optional `args` attribute which is used to pass arguments to the builder executable.

First of all, let's write our `builder.sh` in the current directory:

```
declare -xp
echo foo > $out
```

The command `declare -xp` lists exported variables (`declare` is a builtin bash function). As we covered in the previous pill, Nix computes the output path of the derivation. The resulting `.drv` file contains a list of environment variables passed to the builder. One of these is `$out`.

What we have to do is create something in the path `$out`, be it a file or a directory. In this case we are creating a file.

In addition, we print out the environment variables during the build process. We cannot use env for this, because env is part of coreutils and we don't have a dependency to it yet. We only have bash for now.

Like for coreutils in the previous pill, we get a blessed bash for free from our magic nixpkgs stuff:

```
nix-repl> :l <nixpkgs>
Added 3950 variables.
nix-repl> "${bash}"
"/nix/store/ihmkc7z2wqk3bbipfnlh0yjrlfkkgnv6-bash-4.2-p45"
```

So with the usual trick, we can refer to bin/bash and create our derivation:

```
nix-repl> d = derivation { name = "foo"; builder = "${bash}/bin/bash"; args = [ ./bu
nix-repl> :b d
[1 built, 0.0 MiB DL]

this derivation produced the following outputs:
  out -> /nix/store/gczb4qrag22harvv693wwnflqy7lx5pb-foo
```

We did it! The contents of `/nix/store/w024zci0x1hh1wj6gjq0jagkc1sgrf5r-foo` is really foo. We've built our first derivation.

Note that we used `./builder.sh` and not `"./builder.sh"`. This way, it is parsed as a path, and Nix performs some magic which we will cover later. Try using the string version and you will find that it cannot find `builder.sh`. This is because it tries to find it relative to the temporary build directory.

The builder environment

We can use `nix-store --read-log` to see the logs our builder produced:

```
$ nix-store --read-log /nix/store/gczb4qrag22harvv693wwnflqy7lx5pb-foo
declare -x HOME="/homeless-shelter"
declare -x NIX_BUILD_CORES="4"
declare -x NIX_BUILD_TOP="/tmp/nix-build-foo.drv-0"
declare -x NIX_LOG_FD="2"
declare -x NIX_STORE="/nix/store"
declare -x OLDPWD
declare -x PATH="/path-not-set"
declare -x PWD="/tmp/nix-build-foo.drv-0"
declare -x SHLVL="1"
declare -x TEMP="/tmp/nix-build-foo.drv-0"
declare -x TEMPDIR="/tmp/nix-build-foo.drv-0"
declare -x TMP="/tmp/nix-build-foo.drv-0"
```

```
declare -x TMPDIR="/tmp/nix-build-foo.drv-0"
declare -x builder="/nix/store/q1g0rl8zfmz7r371fp5p42p4acmv297d-bash-4.4-p19/bin/bas
declare -x name="foo"
declare -x out="/nix/store/gczb4qrag22harvv693wwnflqy7lx5pb-foo"
declare -x system="x86_64-linux"
```

Let's inspect those environment variables printed during the build process.

- `$HOME` is not your home directory, and `/homeless-shelter` doesn't exist at all. We force packages not to depend on `$HOME` during the build process.
- `$PATH` plays the same game as `$HOME`
- `$NIX_BUILD_CORES` and `$NIX_STORE` are nix configuration options
- `$PWD` and `$TMP` clearly show that nix created a temporary build directory
- Then `$builder`, `$name`, `$out`, and `$system` are variables set due to the .drv file's contents.

And that's how we were able to use `$out` in our derivation and put stuff in it. It's like Nix reserved a slot in the nix store for us, and we must fill it.

In terms of autotools, `$out` will be the `--prefix` path. Yes, not the make `DESTDIR`, but the `--prefix`. That's the essence of stateless packaging. You don't install the package in a global common path under `/`, you install it in a local isolated path under your nix store slot.

The .drv contents

We added something else to the derivation this time: the args attribute. Let's see how this changed the .drv compared to the previous pill:

```
$ nix derivation show /nix/store/i76pr1cz0za3i9r6xq518bqqvd2raspw-foo.drv
{
  "/nix/store/i76pr1cz0za3i9r6xq518bqqvd2raspw-foo.drv": {
    "outputs": {
      "out": {
        "path": "/nix/store/gczb4qrag22harvv693wwnflqy7lx5pb-foo"
      }
    },
    "inputSrcs": [
      "/nix/store/lb0n38r2b20r8rl1k45a7s4pj6ny22f7-builder.sh"
    ],
    "inputDrvs": {
      "/nix/store/hcgwbx42mcxr7ksnv0i1fg7kw6jvxshb-bash-4.4-p19.drv": [
        "out"
      ]
    },
    "platform": "x86_64-linux",
    "builder": "/nix/store/q1g0rl8zfmz7r371fp5p42p4acmv297d-bash-4.4-p19/bin/bash",
    "args": [
      "/nix/store/lb0n38r2b20r8rl1k45a7s4pj6ny22f7-builder.sh"
    ],
    "env": {
      "builder": "/nix/store/q1g0rl8zfmz7r371fp5p42p4acmv297d-bash-4.4-p19/bin/bash"
      "name": "foo",
```

```
      "out": "/nix/store/gczb4qrag22harvv693wwnflqy7lx5pb-foo",
      "system": "x86_64-linux"
    }
  }
}
```

Much like the usual .drv, except that there's a list of arguments in there passed to the builder (bash) with `builder.sh`... In the nix store..? Nix automatically copies files or directories needed for the build into the store to ensure that they are not changed during the build process and that the deployment is stateless and independent of the building machine. `builder.sh` is not only in the arguments passed to the builder, it's also in the input sources.

Given that `builder.sh` is a plain file, it has no .drv associated with it. The store path is computed based on the filename and on the hash of its contents. Store paths are covered in detail in a later pill.

Packaging a simple C program

Start off by writing a simple C program called `simple.c`:

```c
void main() {
    puts("Simple!");
}
```

And its `simple_builder.sh`:

```bash
export PATH="$coreutils/bin:$gcc/bin"
mkdir $out
gcc -o $out/simple $src
```

Don't worry too much about where those variables come from yet; let's write the derivation and build it:

```
nix-repl> :l <nixpkgs>
nix-repl> simple = derivation { name = "simple"; builder = "${bash}/bin/bash"; args
nix-repl> :b simple
this derivation produced the following outputs:

  out -> /nix/store/ni66p4jfqksbms1616llx3fbs1d232d4-simple
```

Now you can run `/nix/store/ni66p4jfqksbms1616llx3fbs1d232d4-simple/simple` in your shell.

Explanation

We added two new attributes to the derivation call, `gcc` and `coreutils`. In `gcc = gcc;`, the name on the left is the name in the derivation set, and the name on the right refers to the gcc derivation from nixpkgs. The same applies for coreutils.

We also added the `src` attribute, nothing magical — it's just a name, to which the path `./simple.c` is assigned. Like `simple-builder.sh`, `simple.c` will be added to the store.

The trick: every attribute in the set passed to `derivation` will be converted to a string and passed to the builder as an environment variable. This is how the builder gains access to coreutils and gcc: when converted to strings, the derivations evaluate to their output paths, and appending `/bin` to these leads us to their binaries.

The same goes for the `src` variable. `$src` is the path to `simple.c` in the nix store. As an exercise, pretty print the .drv file. You'll see `simple_builder.sh` and `simple.c` listed in the input derivations, along with bash, gcc and coreutils .drv files. The newly added environment variables described above will also appear.

In `simple_builder.sh` we set the `PATH` for gcc and coreutils binaries, so that our build script can find the necessary utilities like mkdir and gcc.

We then create `$out` as a directory and place the binary inside it. Note that gcc is found via the `PATH` environment variable, but it could equivalently be referenced explicitly using `$gcc/bin/gcc`.

Enough of `nix repl`

Drop out of nix repl and write a file `simple.nix`:

```
let
  pkgs = import <nixpkgs> { };
in
derivation {
  name = "simple";
  builder = "${pkgs.bash}/bin/bash";
  args = [ ./simple_builder.sh ];
  gcc = pkgs.gcc;
  coreutils = pkgs.coreutils;
  src = ./simple.c;
  system = builtins.currentSystem;
}
```

Now you can build it with `nix-build simple.nix`. This will create a symlink `result` in the current directory, pointing to the out path of the derivation.

nix-build does two jobs:

- nix-instantiate: parse and evaluate `simple.nix` and return the .drv file corresponding to the parsed derivation set
- `nix-store -r`: realise the .drv file, which actually builds it.

Finally, it creates the symlink.

In the second line of `simple.nix`, we have an `import` function call. Recall that `import` accepts one argument, a nix file to load. In this case, the contents of the file evaluate to a function.

Afterwards, we call the function with the empty set. We saw this already in the fifth pill. To reiterate: `import <nixpkgs> {}` is calling two functions, not one. Reading it as `(import <nixpkgs>) {}` makes this clearer.

The value returned by the nixpkgs function is a set; more specifically, it's a set of derivations. Calling `import <nixpkgs> {}` into a `let`-expression creates the local variable `pkgs` and brings it into scope. This has an effect similar to the `:l <nixpkgs>` we used in nix repl, in that it allows us to easily access derivations such as `bash`, `gcc`, and `coreutils`, but those derivations will have to be explicitly referred to as members of the `pkgs` set (e.g., `pkgs.bash` instead of just `bash`).

Below is a revised version of the `simple.nix` file, using the `inherit` keyword:

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

```
in
derivation {
  name = "simple";
  builder = "${pkgs.bash}/bin/bash";
  args = [ ./simple_builder.sh ];
  inherit (pkgs) gcc coreutils;
  src = ./simple.c;
  system = builtins.currentSystem;
}
```

Here we also take the opportunity to introduce the `inherit` keyword. `inherit foo;` is equivalent to `foo = foo;`. Similarly, `inherit gcc coreutils;` is equivalent to `gcc = gcc; coreutils = coreutils;`. Lastly, `inherit (pkgs) gcc coreutils;` is equivalent to `gcc = pkgs.gcc; coreutils = pkgs.coreutils;`.

This syntax only makes sense inside sets. There's no magic involved, it's simply a convenience to avoid repeating the same name for both the attribute name and the value in scope.

Next pill

We will generalize the builder. You may have noticed that we wrote two separate `builder.sh` scripts in this post. We would like to have a generic builder script instead, especially since each build script goes in the nix store: a bit of a waste.

Is it really that hard to package stuff in Nix? No, here we're studying the fundamentals of Nix.

Generic Builders

Welcome to the 8th Nix pill. In the previous 7th pill we successfully built a derivation. We wrote a builder script that compiled a C file and installed the binary under the nix store.

In this post, we will generalize the builder script, write a Nix expression for GNU hello world and create a wrapper around the derivation built-in function.

Packaging GNU hello world

In the previous pill we packaged a simple .c file, which was being compiled with a raw gcc call. That's not a good example of a project. Many use autotools, and since we're going to generalize our builder, it would be better to do it with the most used build system.

GNU hello world, despite its name, is a simple yet complete project which uses autotools. Fetch the latest tarball here: https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz.

Let's create a builder script for GNU hello world, hello$_{\text{builder}}$.sh:

```
export PATH="$gnutar/bin:$gcc/bin:$gnumake/bin:$coreutils/bin:$gawk/bin:$gzip/bin:$g
tar -xzf $src
cd hello-2.12.1
./configure --prefix=$out
make
make install
```

And the derivation hello.nix:

```
let
  pkgs = import <nixpkgs> { };
in
derivation {
  name = "hello";
  builder = "${pkgs.bash}/bin/bash";
  args = [ ./hello_builder.sh ];
  inherit (pkgs)
    gnutar
    gzip
    gnumake
    gcc
    coreutils
    gawk
    gnused
    gnugrep
    ;
  bintools = pkgs.binutils.bintools;
  src = ./hello-2.12.1.tar.gz;
  system = builtins.currentSystem;
}
```

Nix on darwin

Darwin (i.e. macOS) builds typically use `clang` rather than `gcc` for a C compiler. We can adapt this early example for darwin by using this modified version of `hello.nix`:

```
let
  pkgs = import <nixpkgs> { };
in
derivation {
  name = "hello";
  builder = "${pkgs.bash}/bin/bash";
  args = [ ./hello_builder.sh ];
  inherit (pkgs)
    gnutar
    gzip
    gnumake
    coreutils
    gawk
    gnused
    gnugrep
    ;
  gcc = pkgs.clang;
  bintools = pkgs.clang.bintools.bintools_bin;
  src = ./hello-2.12.1.tar.gz;
  system = builtins.currentSystem;
}
```

Later, we will show how Nix can automatically handle these differences. For now, please be just aware that changes similar to the above may be needed in what follows.

Now build it with `nix-build hello.nix` and you can launch `result/bin/hello`. Nothing easier, but do we have to create a builder.sh for each package? Do we always have to pass the dependencies to the `derivation` function?

Please note the `--prefix=$out` we were talking about in the previous pill.

A generic builder

Let's create a generic `builder.sh` for autotools projects:

```
set -e
unset PATH
for p in $buildInputs; do
    export PATH=$p/bin${PATH:+:}$PATH
done

tar -xf $src

for d in *; do
    if [ -d "$d" ]; then
        cd "$d"
        break
    fi
done

./configure --prefix=$out
make
make install
```

What do we do here?

1. Exit the build on any error with `set -e`.
2. First `unset PATH`, because it's initially set to a non-existent path.
3. We'll see this below in detail, however for each path in `$buildInputs`, we append `bin` to `PATH`.
4. Unpack the source.
5. Find a directory where the source has been unpacked and `cd` into it.
6. Once we're set up, compile and install.

As you can see, there's no reference to "hello" in the builder anymore. It still makes several assumptions, but it's certainly more generic.

Now let's rewrite `hello.nix`:

```
let
  pkgs = import <nixpkgs> { };
in
derivation {
  name = "hello";
  builder = "${pkgs.bash}/bin/bash";
  args = [ ./builder.sh ];
  buildInputs = with pkgs; [
    gnutar
    gzip
    gnumake
    gcc
```

```
    coreutils
    gawk
    gnused
    gnugrep
    binutils.bintools
  ];
  src = ./hello-2.12.1.tar.gz;
  system = builtins.currentSystem;
}
```

All clear, except that buildInputs. However it's easier than any black magic you are thinking of at this moment.

Nix is able to convert a list to a string. It first converts the elements to strings, and then concatenates them separated by a space:

```
nix-repl> builtins.toString 123
"123"
nix-repl> builtins.toString [ 123 456 ]
"123 456"
```

Recall that derivations can be converted to a string, hence:

```
nix-repl> :l <nixpkgs>
Added 3950 variables.
nix-repl> builtins.toString gnugrep
"/nix/store/g5gdylclfh6d224kqh9sja290pk186xd-gnugrep-2.14"
nix-repl> builtins.toString [ gnugrep gnused ]
"/nix/store/g5gdylclfh6d224kqh9sja290pk186xd-gnugrep-2.14 /nix/store/krgdc4sknzpw8iy
```

Simple! The buildInputs variable is a string with out paths separated by space, perfect for bash usage in a for loop.

A more convenient derivation function

We managed to write a builder that can be used for multiple autotools projects. But in the hello.nix expression we are specifying tools that are common to more projects; we don't want to pass them every time.

A natural approach would be to create a function that accepts an attribute set, similar to the one used by the derivation function, and merge it with another attribute set containing values common to many projects.

Create `autotools.nix`:

```
pkgs: attrs:
let
  defaultAttrs = {
    builder = "${pkgs.bash}/bin/bash";
    args = [ ./builder.sh ];
    baseInputs = with pkgs; [
      gnutar
      gzip
      gnumake
      gcc
      coreutils
      gawk
```

```
      gnused
      gnugrep
      binutils.bintools
    ];
    buildInputs = [ ];
    system = builtins.currentSystem;
  };
in
derivation (defaultAttrs // attrs)
```

Ok now we have to remember a little about Nix functions. The whole nix expression of this `autotools.nix` file will evaluate to a function. This function accepts a parameter `pkgs`, then returns a function which accepts a parameter `attrs`.

The body of the function is simple, yet at first sight it might be hard to grasp:

1. First drop in the scope the magic `pkgs` attribute set.
2. Within a let expression we define a helper variable, `defaultAttrs`, which serves as a set of common attributes used in derivations.
3. Finally we create the derivation with that strange expression, (`defaultAttrs // attrs`).

The // operator is an operator between two sets. The result is the union of the two sets. In case of conflicts between attribute names, the value on the right set is preferred.

So we use `defaultAttrs` as base set, and add (or override) the attributes from `attrs`.

A couple of examples ought to be enough to clear out the behavior of the operator:

```
nix-repl> { a = "b"; } // { c = "d"; }
{ a = "b"; c = "d"; }
nix-repl> { a = "b"; } // { a = "c"; }
{ a = "c"; }
```

Exercise: Complete the new `builder.sh` by adding `$baseInputs` in the `for` loop together with `$buildInputs`. As you noticed, we passed that new variable in the derivation. Instead of merging buildInputs with the base ones, we prefer to preserve buildInputs as seen by the caller, so we keep them separated. Just a matter of choice.

Then we rewrite `hello.nix` as follows:

```
let
  pkgs = import <nixpkgs> { };
  mkDerivation = import ./autotools.nix pkgs;
in
mkDerivation {
  name = "hello";
  src = ./hello-2.12.1.tar.gz;
}
```

Finally! We got a very simple description of a package! Below are a couple of remarks that you may find useful as you're continuing to understand the nix language:

- We assigned to pkgs the import that we did in the previous expressions in the "with". Don't be afraid, it's that straightforward.
- The mkDerivation variable is a nice example of partial application, look at it as (`import ./autotools.nix`) `pkgs`. First we import the expression, then we apply the

pkgs parameter. That will give us a function that accepts the attribute set attrs.

- We create the derivation specifying only name and src. If the project eventually needed other dependencies to be in PATH, then we would simply add those to buildInputs (not specified in hello.nix because empty).

Note we didn't use any other library. Special C flags may be needed to find include files of other libraries at compile time, and ld flags at link time.

Conclusion

Nix gives us the bare metal tools for creating derivations, setting up a build environment and storing the result in the nix store.

Out of this pill we managed to create a generic builder for autotools projects, and a function mkDerivation that composes by default the common components used in autotools projects instead of repeating them in all the packages we would write.

We are familiarizing ourselves with the way a Nix system grows up: it's about creating and composing derivations with the Nix language.

Analogy: in C you create objects in the heap, and then you compose them inside new objects. Pointers are used to refer to other objects.

In Nix you create derivations stored in the nix store, and then you compose them by creating new derivations. Store paths are used to refer to other derivations.

Next pill

...we will talk a little about runtime dependencies. Is the GNU hello world package self–contained? What are its runtime dependencies? We only specified build dependencies by means of using other derivations in the "hello" derivation.

Automatic Runtime Dependencies

Welcome to the 9th Nix pill. In the previous 8th pill we wrote a generic builder for autotools projects. We fed in build dependencies and a source tarball, and we received a Nix derivation as a result.

Today we stop by the GNU hello program to analyze build and runtime dependencies, and we enhance our builder to eliminate unnecessary runtime dependencies.

Build dependencies

Let's start analyzing build dependencies for our GNU hello package:

```
$ nix-instantiate hello.nix
/nix/store/z77vn965a59irqnrrjvbspiyl2rph0jp-hello.drv
$ nix-store -q --references /nix/store/z77vn965a59irqnrrjvbspiyl2rph0jp-hello.drv
/nix/store/0q6pfasdma4as22kyaknk4kwx4h58480-hello-2.10.tar.gz
/nix/store/1zcs1y4n27lqs0gw4v038i303pb89rw6-coreutils-8.21.drv
/nix/store/2h4b30hlfw4fhqx10wwi71mpim4wr877-gnused-4.2.2.drv
/nix/store/39bgdjissw9gyi4y5j9wanf4dbjpbl07-gnutar-1.27.1.drv
/nix/store/7qa70nay0if4x291rsjr7h9lfl6pl7b1-builder.sh
/nix/store/g6a0shr58qvx2vi6815acgp9lnfh9yy8-gnugrep-2.14.drv
/nix/store/jdggv3q1sb15140qdx0apvyrps41m4lr-bash-4.2-p45.drv
/nix/store/pglhiyp1zdbmax4cglkpz98nspfgbnwr-gnumake-3.82.drv
/nix/store/q9l257jn9lndbi3r9ksnvf4dr8cwxzk7-gawk-4.1.0.drv
```

```
/nix/store/rgyrqxz1ilv90r01zxl0sq5nq0cq7v3v-binutils-2.23.1.drv
/nix/store/qzxhby795niy6wlagfpbja27dgsz43xk-gcc-wrapper-4.8.3.drv
/nix/store/sk590g7fv53m3zp0ycnxsc41snc2kdhp-gzip-1.6.drv
```

It has precisely the derivations referenced in the `derivation` function; nothing more, nothing less. Of course, we may not use some of them at all. However, given that our generic `mkDerivation` function always pulls such dependencies (think of it like build-essential from Debian), we will already have these packages in the nix store for any future packages that need them.

Why are we looking at `.drv` files? Because the `hello.drv` file is the representation of the build action that builds the `hello` out path. As such, it contains the input derivations needed before building `hello`.

Digression about NAR files

The `NAR` format is the "Nix ARchive". This format was designed due to existing archive formats, such as `tar`, being insufficient. Nix benefits from deterministic build tools, but commonly used archivers lack this property: they add padding, they do not sort files, they add timestamps, and so on. This can result in directories containing bit-identical files turning into non-bit-identical archives, which leads to different hashes.

Thus the `NAR` format was developed as a simple, deterministic archive format. =NAR=s are used extensively within Nix, as we will see below.

For more rationale and implementation details behind `NAR` see Dolstra's PhD Thesis.

To create NAR archives from store paths, we can use `nix-store --dump` and `nix-store --restore`.

Runtime dependencies

We now note that Nix automatically recognized build dependencies once our `derivation` call referred to them, but we never specified the runtime dependencies.

Nix handles runtime dependencies for us automatically. The technique it uses to do so may seem fragile at first glance, but it works so well that the NixOS operating system is built off of it. The underlying mechanism relies on the hash of the store paths. It proceeds in three steps:

1. Dump the derivation as a NAR. Recall that this is a serialization of the derivation output – meaning this works fine whether the output is a single file or a directory.
2. For each build dependency `.drv` and its relative out path, search the contents of the NAR for this out path.
3. If the path is found, then it's a runtime dependency.

The snippet below shows the dependencies for `hello`.

```
$ nix-instantiate hello.nix
/nix/store/z77vn965a59irqnrrjvbspiyl2rph0jp-hello.drv
$ nix-store -r /nix/store/z77vn965a59irqnrrjvbspiyl2rph0jp-hello.drv
/nix/store/a42k52zwv6idmf50r9lps1nzwq9khvpf-hello
$ nix-store -q --references /nix/store/a42k52zwv6idmf50r9lps1nzwq9khvpf-hello
/nix/store/94n64qy99ja0vgbkf675nyk39g9b978n-glibc-2.19
/nix/store/8jm0wksask7cpf85miyakihyfch1y21q-gcc-4.8.3
/nix/store/a42k52zwv6idmf50r9lps1nzwq9khvpf-hello
```

We see that `glibc` and `gcc` are runtime dependencies. Intuitively, `gcc` shouldn't be in this list! Displaying the printable strings in the `hello` binary shows that the out path of `gcc` does indeed appear:

```
$ strings result/bin/hello|grep gcc
/nix/store/94n64qy99ja0vgbkf675nyk39g9b978n-glibc-2.19/lib:/nix/store/8jm0wksask7cpf
```

This is why Nix added `gcc`. But why is that path present in the first place? The answer is that it is the ld rpath: the list of directories where libraries can be found at runtime. In other distributions, this is usually not abused. But in Nix, we have to refer to particular versions of libraries, and thus the rpath has an important role.

The build process adds the `gcc` lib path thinking it may be useful at runtime, but this isn't necessary. To address issues like these, Nix provides a tool called patchelf, which reduces the rpath to the paths that are actually used by the binary.

Even after reducing the rpath, the `hello` binary would still depend upon `gcc` because of some debugging information. This unnecessarily increases the size of our runtime dependencies. We'll explore how `strip` can help us with that in the next section.

Another phase in the builder

We will add a new phase to our autotools builder. The builder has six phases already:

1. The "environment setup" phase
2. The "unpack phase": we unpack the sources in the current directory (remember, Nix changes to a temporary directory first)
3. The "change directory" phase, where we change source root to the directory that has been unpacked
4. The "configure" phase: `./configure`
5. The "build" phase: `make`
6. The "install" phase: `make install`

Now we will add a new phase after the installation phase, which we call the "fixup" phase. At the end of the `builder.sh`, we append:

```
find $out -type f -exec patchelf --shrink-rpath '{}' \; -exec strip '{}' \; 2>/dev/n
```

That is, for each file we run `patchelf --shrink-rpath` and `strip`. Note that we used two new commands here, `find` and `patchelf`. These must be added to our derivation.

Exercise: Add `findutils` and `patchelf` to the `baseInputs` of `autotools.nix`.

Now, we rebuild `hello.nix`...

```
$ nix-build hello.nix
[...]
$ nix-store -q --references result
/nix/store/94n64qy99ja0vgbkf675nyk39g9b978n-glibc-2.19
/nix/store/md4a3zv0ipqzsybhjb8ndjhhga1dj88x-hello
```

and we see that `glibc` is a runtime dependency but `gcc` is not there anymore. This is exactly what we wanted.

The package is self-contained. This means that we can copy its closure onto another machine and we will be able to run it. Remember, only a very few components under the `/nix/store` are required to run nix. The `hello` binary will use the exact version of glibc library and interpreter referred to in the binary, rather than the system one:

```
$ ldd result/bin/hello
 linux-vdso.so.1 (0x00007fff11294000)
 libc.so.6 => /nix/store/94n64qy99ja0vgbkf675nyk39g9b978n-glibc-2.19/lib/libc.so.6 (
 /nix/store/94n64qy99ja0vgbkf675nyk39g9b978n-glibc-2.19/lib/ld-linux-x86-64.so.2 (0x
```

Of course, the executable will run fine as long as everything is under the /nix/store path.

Conclusion

We saw some of the tools Nix provides, along with their features. In particular, we saw how Nix is able to compute runtime dependencies automatically. This is not limited to only shared libraries, but can also reference executables, scripts, Python libraries, and so forth.

Approaching builds in this way makes packages self–contained, ensuring (apart from data and configuration) that copying the runtime closure onto another machine is sufficient to run the program. This enables us to run programs without installation using nix-shell, and forms the basis for reliable deployment in the cloud.

Next pill

The next pill will introduce nix-shell. With nix-build, we've always built derivations from scratch: the source gets unpacked, configured, built, and installed. But this can take a long time for large packages. What if we want to apply some small changes and compile incrementally instead, yet still want to keep a self–contained environment similar to nix-build? nix-shell enables this.

Developing with nix-shell

Welcome to the 10th Nix pill. In the previous 9th pill we saw one of the powerful features of Nix: automatic discovery of runtime dependencies. We also finalized the GNU hello package.

In this pill, we will introduce the nix-shell tool and use it to hack on the GNU hello program. We will see how nix-shell gives us an isolated environment while we modify the source files of the project, similar to how nix-build gave us an isolated environment while building the derivation.

Finally, we will modify our builder to work more ergonomically with a nix-shell–focused workflow.

What is nix-shell?

The nix–shell tool drops us in a shell after setting up the environment variables necessary to hack on a derivation. It does not build the derivation; it only serves as a preparation so that we can run the build steps manually.

Recall that in a nix environment, we don't have access to libraries or programs unless they have been installed with nix-env. However, installing libraries with nix-env is not good practice. We prefer to have isolated environments for development, which nix-shell provides for us.

We can call nix-shell on any Nix expression which returns a derivation, but the resulting bash shell's PATH does not have the utilities we want:

```
$ nix-shell hello.nix
[nix-shell]$ make
bash: make: command not found
[nix-shell]$ echo $baseInputs
/nix/store/jff4a6zqi0yrladx3kwy4v6844s3swpc-gnutar-1.27.1 [...]
```

This shell is rather useless. It would be reasonable to expect that the GNU `hello` build inputs are available in `PATH`, including GNU `make`, but this is not the case.

However, we do have the environment variables that we set in the derivation, like `$baseInputs`, `$buildInputs`, `$src`, and so on.

This means that we can `source` our `builder.sh`, and it will build the derivation. You may get an error in the installation phase, because your user may not have the permission to write to `/nix/store`:

```
[nix-shell]$ source builder.sh
...
```

The derivation didn't install, but it did build. Note the following:

- We sourced `builder.sh` and it ran all of the build steps, including setting up the `PATH` for us.
- The working directory is no longer a temp directory created by `nix-build`, but is instead the directory in which we entered the shell. Therefore, `hello-2.10` has been unpacked in the current directory.

We are able to `cd` into `hello-2.10` and type `make`, because `make` is now available.

The take–away is that `nix-shell` drops us in a shell with the same (or very similar) environment used to run the builder.

A builder for nix–shell

The previous steps require some manual commands to be run and are not optimized for a workflow centered on `nix-shell`. We will now improve our builder to be more `nix-shell` friendly.

There are a few things that we would like to change.

First, when we `source=d the =builder.sh` file, we obtained the file in the current directory. What we really wanted was the `builder.sh` that is stored in the nix store, as this is the file that would be used by `nix-build`. To achieve this, the correct technique is to pass an environment variable through the derivation. (Note that `$builder` is already defined, but it points to the bash executable rather than our `builder.sh`. Our `builder.sh` is passed as an argument to bash.)

Second, we don't want to run the whole builder: we only want to setup the necessary environment for manually building the project. Thus, we can break `builder.sh` into two files: a `setup.sh` for setting up the environment, and the real `builder.sh` that `nix-build` expects.

During our refactoring, we will wrap the build phases in functions to give more structure to our design. Additionally, we'll move the `set -e` to the builder file instead of the setup file. The `set -e` is annoying in `nix-shell`, as it will terminate the shell if an error is encountered (such as a mistyped command.)

Here is our modified `autotools.nix`. Noteworthy is the `setup = ./setup.sh;` attribute in the derivation, which adds `setup.sh` to the nix store and correspondingly adds a `$setup` environment variable in the builder.

```
pkgs: attrs:
let
  defaultAttrs = {
    builder = "${pkgs.bash}/bin/bash";
    args = [ ./builder.sh ];
    setup = ./setup.sh;
    baseInputs = with pkgs; [
      gnutar
      gzip
      gnumake
      gcc
      coreutils
      gawk
      gnused
      gnugrep
      binutils.bintools
      patchelf
      findutils
    ];
    buildInputs = [ ];
    system = builtins.currentSystem;
  };
in
derivation (defaultAttrs // attrs)
```

Thanks to that, we can split `builder.sh` into `setup.sh` and `builder.sh`. What `builder.sh` does is source `$setup` and call the `genericBuild` function. Everything else is just some changes to the bash script.

Here is the modified `builder.sh`:

```
set -e
source $setup
genericBuild
```

Here is the newly added `setup.sh`:

```
unset PATH
for p in $baseInputs $buildInputs; do
    export PATH=$p/bin${PATH:+:}$PATH
done

function unpackPhase() {
    tar -xzf $src

    for d in *; do
    if [ -d "$d" ]; then
        cd "$d"
        break
    fi
    done
}
```

```
function configurePhase() {
    ./configure --prefix=$out
}

function buildPhase() {
    make
}

function installPhase() {
    make install
}

function fixupPhase() {
    find $out -type f -exec patchelf --shrink-rpath '{}' \; -exec strip '{}' \; 2>/d
}

function genericBuild() {
    unpackPhase
    configurePhase
    buildPhase
    installPhase
    fixupPhase
}
```

Finally, here is `hello.nix`:

```
let
  pkgs = import <nixpkgs> { };
  mkDerivation = import ./autotools.nix pkgs;
in
mkDerivation {
  name = "hello";
  src = ./hello-2.12.1.tar.gz;
}
```

Now back to nix–shell:

```
$ nix-shell hello.nix
[nix-shell]$ source $setup
[nix-shell]$
```

Now, for example, you can run `unpackPhase` which unpacks `$src` and enters the directory. And you can run commands like `./configure`, `make`, and so forth manually, or run phases with their respective functions.

The process is that straightforward. `nix-shell` builds the `.drv` file and its input dependencies, then drops into a shell by setting up the environment variables necessary to build the `.drv`. In particular, the environment variables in the shell match those passed to the `derivation` function.

Conclusion

With `nix-shell` we are able to drop into an isolated environment suitable for developing a project. This environment provides the necessary dependencies for the development shell, similar to how `nix-build` provides the necessary dependencies to a builder. Additionally, we can build and debug the project manually, executing step–by–step like we would in any other operating system. Note that we never installed tools such gcc or make system–wide; these tools and libraries are isolated and available per–build.

Next pill

In the next pill, we will clean up the nix store. We have written and built derivations which add to the nix store, but until now we haven't worried about cleaning up the used space in the store.

The Garbage Collector

Welcome to the 11th Nix pill. In the previous 10th pill, we drew a parallel between the isolated build environment provided by nix-build and the isolated development shell provided by nix-shell. Using nix-shell allowed us to debug, modify, and manually build software using an environment that is almost identical to the one provided by nix-build.

Today, we will stop focusing on packaging and instead look at a critical component of Nix: the garbage collector. When we use Nix tools, we are often building derivations. This includes .drv files as well as out paths. These artifacts go in the Nix store and take up space in our storage. Eventually we may wish to free up some space by removing derivations we no longer need. This is the focus of the 11th pill. By default, Nix takes a relatively conservative approach when automatically deciding which derivations are "needed". In this pill, we will also see a technique to conduct more destructive upgrade and deletion operations.

How does garbage collection work?

Programming languages with garbage collectors use the concept of a set of "garbage collector (or 'GC') roots" to keep track of "live" objects. A GC root is an object that is always considered "live" (unless explicitly removed as GC root). The garbage collection process starts from the GC roots and proceeds by recursively marking object references as "live". All other objects can be collected and deleted.

Instead of objects, Nix's garbage collection operates on store paths, with the GC roots themselves being store paths. . This approach is much more principled than traditional package managers such as dpkg or rpm, which may leave around unused packages or dangling files.

The implementation is very simple and transparent to the user. The primary GC roots are stored under /nix/var/nix/gcroots. If there is a symlink to a store path, then the linked store path is a GC root.

Nix allows this directory to have subdirectories: it will simply recursively traverse the subdirectories in search of symlinks to store paths. When a symlink is encountered, its target is added to the list of live store paths.

In summary, Nix maintains a list of GC roots. These roots can then be used to compute a list of all live store paths. Any other store paths are considered dead. Deleting these paths is now straightforward. Nix first moves dead store paths to /nix/store/trash, which is an atomic operation. Afterwards, the trash is emptied.

Playing with the GC

Before we begin we first run the nix garbage collector so that we have a clean setup for our experiments:

```
$ nix-collect-garbage
finding garbage collector roots...
[...]
deleting unused links...
note: currently hard linking saves -0.00 MiB
1169 store paths deleted, 228.43 MiB freed
```

If we run the garbage collector again it won't find anything new to delete, as we expect. After running the garbage collector, the nix store only contains paths with references from the GC roots.

We now install a new program, `bsd-games`, inspect its store path, and examine its GC root. The `nix-store -q --roots` command is used to query the GC roots that refer to a given derivation. In this case, our current user environment refers to `bsd-games`:

```
$ nix-env -iA nixpkgs.bsdgames
$ readlink -f `which fortune`
/nix/store/b3lxx3d3ggxcggvjw5n0m1ya1gcrmbyn-bsd-games-2.17/bin/fortune
$ nix-store -q --roots `which fortune`
/nix/var/nix/profiles/default-9-link
$ nix-env --list-generations
[...]
   9   2014-08-20 12:44:14   (current)
```

Now we remove it and run the garbage collector, and note that `bsd-games` is still in the nix store:

```
$ nix-env -e bsd-games
uninstalling `bsd-games-2.17'
$ nix-collect-garbage
[...]
$ ls /nix/store/b3lxx3d3ggxcggvjw5n0m1ya1gcrmbyn-bsd-games-2.17
bin  share
```

The old generation is still in the nix store because it is a GC root. As we will see below, all profiles and their generations are automatically GC roots.

Removing a GC root is simple. In our case, we delete the generation that refers to `bsd-games`, run the garbage collector, and note that `bsd-games` is no longer in the nix store:

```
$ rm /nix/var/nix/profiles/default-9-link
$ nix-env --list-generations
[...]
   8   2014-07-28 10:23:24
  10   2014-08-20 12:47:16   (current)
$ nix-collect-garbage
[...]
$ ls /nix/store/b3lxx3d3ggxcggvjw5n0m1ya1gcrmbyn-bsd-games-2.17
ls: cannot access /nix/store/b3lxx3d3ggxcggvjw5n0m1ya1gcrmbyn-bsd-games-2.17: No suc
```

Note: `nix-env --list-generations` does not rely on any particular metadata. It is able to list generations based solely on the file names under the profiles directory.

Note that we removed the link from `/nix/var/nix/profiles`, not from `/nix/var/nix/gcroots`. In addition to the latter, Nix treats `/nix/var/nix/profiles` as a GC root. This is useful because it means that any profile and its generations are GC roots. Other paths are considered GC roots as well; for example, `/run/booted-system` on NixOS. The command `nix-store --gc --print-roots` prints all paths considered as GC roots when running the garbage collector.

Indirect roots

Recall that building the GNU `hello` package with `nix-build` produces a `result` symlink in the current directory. Despite the garbage collection done above, the `hello` program is still working. Therefore, it has not been garbage collected. Since there is no other derivation that depends upon the GNU `hello` package, it must be a GC root.

In fact, `nix-build` automatically adds the `result` symlink as a GC root. Note that this is not the built derivation, but the symlink itself. These GC roots are added under `/nix/var/nix/gcroots/auto`.

```
$ ls -l /nix/var/nix/gcroots/auto/
total 8
drwxr-xr-x 2 nix nix 4096 Aug 20 10:24 ./
drwxr-xr-x 3 nix nix 4096 Jul 24 10:38 ../
lrwxrwxrwx 1 nix nix   16 Jul 31 10:51 xlgz5x2ppa0m72z5qfc78b8wlciwvgiz -> /home/nix
```

The name of the GC root symlink is not important to us at this time. What is important is that such a symlink exists and points to `/home/nix/result`. This is called an **indirect GC root**. A GC root is considered indirect if its specification is outside of `/nix/var/nix/gcroots`. In this case, this means that the target of the `result` symlink will not be garbage collected.

To remove a derivation considered "live" by an indirect GC root, there are two possibilities:

- Remove the indirect GC root from `/nix/var/nix/gcroots/auto`.
- Remove the `result` symlink.

In the first case, the derivation will be deleted from the nix store during garbage collection, and `result` becomes a dangling symlink. In the second case, the derivation is removed as well as the indirect root in `/nix/var/nix/gcroots/auto`.

Running `nix-collect-garbage` after deleting the GC root or the indirect GC root will remove the derivation from the store.

Cleanup everything

The main source of software duplication in the nix store comes from GC roots, due to `nix-build` and profile generations. Running `nix-build` results in a GC root for the build that refers to a specific version of specific libraries, such as `glibc`. After an upgrade, we must delete the previous build if we want the garbage collector to remove the corresponding derivation, as well as if we want old dependencies cleaned up.

The same holds for profiles. Manipulating the `nix-env` profile will create further generations. Old generations refer to old software, thus increasing duplication in the nix store after an upgrade.

Other systems typically "forget" everything about their previous state after an upgrade. With Nix, we can perform this type of upgrade (having Nix remove all old derivations, including old generations), but we do so manually. There are four steps to doing this:

- First, we download a new version of the nixpkgs channel, which holds the description of all the software. This is done via `nix-channel --update`.

- Then we upgrade our installed packages with `nix-env -u`. This will bring us into a new generation with updated software.
- Then we remove all the indirect roots generated by `nix-build`: beware, as this will result in dangling symlinks. A smarter strategy would also remove the target of those symlinks.
- Finally, the `-d` option of `nix-collect-garbage` is used to delete old generations of all profiles, then collect garbage. After this, you lose the ability to rollback to any previous generation. It is important to ensure the new generation is working well before running this command.

The four steps are shown below:

```
$ nix-channel --update
$ nix-env -u --always
$ rm /nix/var/nix/gcroots/auto/*
$ nix-collect-garbage -d
```

Conclusion

Garbage collection in Nix is a powerful mechanism to clean up your system. The `nix-store` commands allow us to know why a certain derivation is present in the nix store, and whether or not it is eligible for garbage collection. We also saw how to conduct more destructive deletion and upgrade operations.

Next pill

In the next pill, we will package another project and introduce the "inputs" design pattern. We've only played with a single derivation until now; however we'd like to start organizing a small repository of software. The "inputs" pattern is widely used in nixpkgs; it allows us to decouple derivations from the repository itself and increase customization opportunities.

Package Repositories and the Inputs Design Pattern

Welcome to the 12th Nix pill. In the previous 11th pill, we stopped packaging and cleaned up the system with the garbage collector.

This time, we will resume packaging and improve different aspects of it. We will also demonstrate how to create a repository of multiple packages.

Repositories in Nix

Package repositories in Nix arose naturally from the need to organize packages. There is no preset directory structure or packaging policy prescribed by Nix itself; Nix, as a full, functional programming language, is powerful enough to support multiple different repository formats.

Over time, the `nixpkgs` repository evolved a particular structure. This structure reflects the history of Nix as well as the design patterns adopted by its users as useful tools in building and organizing packages. Below, we will examine some of these patterns in detail.

The single repository pattern

Different operating system distributions have different opinions about how package repositories should be organized. Systems like Debian scatter packages in several small repositories (which tends to make tracking interdependent changes more difficult, and hinders contributions to the repositories), while systems like Gentoo put all package descriptions in a single repository.

Nix follows the "single repository" pattern by placing all descriptions of all packages into nixpkgs. This approach has proven natural and attractive for new contributions.

For the rest of this pill, we will adopt the single repository pattern. The natural implementation in Nix is to create a top−level Nix expression, followed by one expression for each package. The top−level expression imports and combines all package expressions in an attribute set mapping names to packages.

In some programming languages, such an approach – including every possible package description in a single data structure – would be untenable due to the language needing to load the entire data structure into memory before operating on it. Nix, however, is a lazy language and only evaluates what is needed.

Packaging `graphviz`

We have already packaged GNU `hello`. Next, we will package a graph−drawing program called `graphviz` so that we can create a repository containing multiple packages. The `graphviz` package was selected because it uses the standard autotools build system and requires no patching. It also has optional dependencies, which will give us an opportunity to illustrate a technique to configure builds to a particular situation.

First, we download `graphviz` from gitlab. The `graphviz.nix` expression is straightforward:

```
let
  pkgs = import <nixpkgs> { };
  mkDerivation = import ./autotools.nix pkgs;
in
mkDerivation {
  name = "graphviz";
  src = ./graphviz-2.49.3.tar.gz;
}
```

If we build the project with `nix-build graphviz.nix`, we will get runnable binaries under `result/bin`. Notice how we reused the same `autotools.nix` of `hello.nix`.

By default, `graphviz` does not compile with the ability to produce `png` files. Thus, the derivation above will build a binary supporting only the native output formats, as we see below:

```
$ echo 'graph test { a -- b }'|result/bin/dot -Tpng -o test.png
Format: "png" not recognized. Use one of: canon cmap [...]
```

If we want to produce a `png` file with `graphviz`, we must add it to our derivation. The place to do so is in `autotools.nix`, where we created a `buildInputs` variable that gets concatenated to `baseInputs`. This is the exact reason for this variable: to allow users of `autotools.nix` to add additional inputs from package expressions.

Version 2.49 of `graphviz` has several plugins to output `png`. For simplicity, we will use `libgd`.

Passing library information to `pkg-config` via environment variables

The `graphviz` configuration script uses `pkg-config` to specify which flags are passed to the compiler. Since there is no global location for libraries, we need to tell `pkg-config` where to find its description files, which tell the configuration script where to find headers and libraries.

In classic POSIX systems, `pkg-config` just finds the `.pc` files of all installed libraries in system folders like `/usr/lib/pkgconfig`. However, these files are not present in the isolated environments presented to Nix.

As an alternative, we can inform `pkg-config` about the location of libraries via the `PKG_CONFIG_PATH` environment variable. We can populate this environment variable using the same trick we used for `PATH`: automatically filling the variables from `buildInputs`. This is the relevant snippet of `setup.sh`:

```
for p in $baseInputs $buildInputs; do
    if [ -d $p/bin ]; then
        export PATH="$p/bin${PATH:+:}$PATH"
    fi
    if [ -d $p/lib/pkgconfig ]; then
        export PKG_CONFIG_PATH="$p/lib/pkgconfig${PKG_CONFIG_PATH:+:}$PKG_CONFIG_PAT
    fi
done
```

Now if we add derivations to `buildInputs`, their `lib/pkgconfig` and `bin` paths are automatically added in `setup.sh`.

Completing graphviz with **`gd`**

Below, we finish the expression for `graphviz` with `gd` support. Note the use of the `with` expression in `buildInputs` to avoid repeating `pkgs`:

```
let
  pkgs = import <nixpkgs> { };
  mkDerivation = import ./autotools.nix pkgs;
in
mkDerivation {
  name = "graphviz";
  src = ./graphviz-2.49.3.tar.gz;
  buildInputs = with pkgs; [
    pkg-config
    (pkgs.lib.getLib gd)
    (pkgs.lib.getDev gd)
  ];
}
```

We add `pkg-config` to the derivation to make this tool available for the configure script. As `gd` is a package with split outputs, we need to add both the library and development outputs.

After building, `graphviz` can now create =png=s.

The repository expression

Now that we have two packages, we want to combine them into a single repository. To do so, we'll mimic what `nixpkgs` does: we will create a single attribute set containing derivations. This attribute set can then be imported, and derivations can be selected by accessing the top–level attribute set.

Using this technique we are able to abstract from the file names. Instead of referring to a package by `REPO/some/sub/dir/package.nix`, this technique allows us to select a derivation as `importedRepo.package` (or `pkgs.package` in our examples).

To begin, create a default.nix in the current directory:

```
{
  hello = import ./hello.nix;
  graphviz = import ./graphviz.nix;
}
```

This file is ready to use with `nix repl`:

```
$ nix repl
nix-repl> :l default.nix
Added 2 variables.
nix-repl> hello
«derivation /nix/store/dkib02g54fpdqgpskswgp6m7bd7mgx89-hello.drv»
nix-repl> graphviz
«derivation /nix/store/zqv520v9mk13is0w980c91z7q1vkhhil-graphviz.drv»
```

With `nix-build`, we can pass the −A option to access an attribute of the set from the given `.nix` expression:

```
$ nix-build default.nix -A hello
[...]
$ result/bin/hello
Hello, world!
```

The `default.nix` file is special. When a directory contains a `default.nix` file, it is used as the implicit nix expression of the directory. This, for example, allows us to run `nix-build -A hello` without specifying `default.nix` explicitly.

We can now use `nix-env` to install the package into our user environment:

```
$ nix-env -f . -iA graphviz
[...]
$ dot -V
```

Taking a closer look at the above command, we see the following options:

- The −f option is used to specify the expression to use. In this case, the expression is the `./default.nix` of the current directory.
- The −i option stands for "installation".
- The −A is the same as above for `nix-build`.

We reproduced the very basic behavior of `nixpkgs`: combining multiple derivations into a single, top−level attribute set.

The inputs pattern

The approach we've taken so far has a few problems:

- First, `hello.nix` and `graphviz.nix` are dependent on `nixpkgs`, which they import directly. A better approach would be to pass in `nixpkgs` as an argument, as we did in `autotools.nix`.
- Second, we don't have a straightforward way to compile different variants of the same software, such as `graphviz` with or without `libgd` support.

- Third, we don't have a way to test `graphviz` with a particular `libgd` version.

Until now, our approach to addressing the above problems has been inadequate and required changing the nix expression to match our needs. With the `inputs` pattern, we provide another answer: let the user change the `inputs` of the expression.

When we talk about "the inputs of an expression", we are referring to the set of derivations needed to build that expression. In this case:

- `mkDerivation` from `autotools`. Recall that `mkDerivation` has an implicit dependency on the toolchain.
- `libgd` and its dependencies.

The `./src` directory is also an input, but we wouldn't change the source from the caller. In `nixpkgs` we prefer to write another expression for version bumps (e.g. because patches or different inputs are needed).

Our goal is to make package expressions independent of the repository. To achieve this, we use functions to declare inputs for a derivation. For example, with `graphviz.nix`, we make the following changes to make the derivation independent of the repository and customizable:

```
{ mkDerivation, lib, gdSupport ? true, gd, pkg-config }:

mkDerivation {
  name = "graphviz";
  src = ./graphviz-2.49.3.tar.gz;
  buildInputs =
    if gdSupport
      then [
        pkg-config
        (lib.getLib gd)
        (lib.getDev gd)
      ]
      else [];
}
```

Recall that "`{...}: ...`" is the syntax for defining functions accepting an attribute set as argument; the above snippet just defines a function.

We made `gd` and its dependencies optional. If `gdSupport` is true (which it is by default), we will fill `buildInputs` and `graphviz` will be built with `gd` support. Otherwise, if an attribute set is passed with `gdSupport = false;`, the build will be completed without `gd` support.

Going back to `default.nix`, we modify our expression to utilize the inputs pattern:

```
let
  pkgs = import <nixpkgs> { };
  mkDerivation = import ./autotools.nix pkgs;
in
with pkgs;
{
  hello = import ./hello.nix { inherit mkDerivation; };
  graphviz = import ./graphviz.nix {
    inherit
      mkDerivation
```

```
      lib
      gd
      pkg-config
      ;
  };
  graphvizCore = import ./graphviz.nix {
    inherit
      mkDerivation
      lib
      gd
      pkg-config
      ;
    gdSupport = false;
  };
}
```

We factorized the import of `nixpkgs` and `mkDerivation`, and also added a variant of `graphviz` with `gd` support disabled. The result is that both `hello.nix` (left as an exercise for the reader) and `graphviz.nix` are independent of the repository and customizable by passing specific inputs.

If we wanted to build `graphviz` with a specific version of `gd`, it would suffice to pass `gd = ...;`.

If we wanted to change the toolchain, we would simply pass a different `mkDerivation` function.

Let's take a closer look at the snippet and dissect the syntax:

- The entire expression in `default.nix` returns an attribute set with the keys `hello`, `graphviz`, and `graphvizCore`.
- With "`let`", we define some local variables.
- We bring `pkgs` into the scope when defining the package set. This saves us from having to type `pkgs`" repeatedly.
- We import `hello.nix` and `graphviz.nix`, which each return a function. We call the functions with a set of inputs to get back the derivation.
- The "`inherit x`" syntax is equivalent to "`x = x`". This means that the "`inherit gd`" here, combined with the above "`with pkgs;`", is equivalent to "`gd = pkgs.gd`".

The entire repository of this can be found at the pill 12 gist.

Conclusion

The "`inputs`" pattern allows our expressions to be easily customizable through a set of arguments. These arguments could be flags, derivations, or any other customizations enabled by the nix language. Our package expressions are simply functions: there is no extra magic present.

The "`inputs`" pattern also makes the expressions independent of the repository. Given that we pass all needed information through arguments, it is possible to use these expressions in any other context.

Next pill

In the next pill, we will talk about the "callPackage" design pattern. This removes the tedium of specifying the names of the inputs twice: once in the top−level default.nix, and once in the package expression. With callPackage, we will implicitly pass the necessary inputs from the top−level expression.

Callpackage Design Pattern

Welcome to the 13th Nix pill. In the previous 12th pill, we introduced the first basic design pattern for organizing a repository of software. In addition, we packaged graphviz so that we had two packages to bundle into an example repository.

The next design pattern we will examine is called the callPackage pattern. This technique is extensively used in nixpkgs, and it's the current de facto standard for importing packages in a repository. Its purpose is to reduce the duplication of identifiers between package derivation inputs and repository derivations.

The callPackage convenience

In the previous pill, we demonstrated how the inputs pattern decouples packages from the repository. This allowed us to manually pass the inputs to the derivation; the derivation declares its inputs, and the caller passes the arguments.

However, as with usual programming languages, there is some duplication of work: we declare parameter names and then we pass arguments, typically with the same name. For example, if we define a package derivation using the inputs pattern such as:

```
{ input1, input2, ... }:
...
```

We would likely want to bundle that package derivation into a repository via an attribute set defined as something like:

```
rec {
  lib1 = import package1.nix { inherit input1 input2; };
  program2 = import package2.nix { inherit inputX inputY lib1; };
}
```

There are two things to note. First, that inputs often have the same name as attributes in the repository itself. Second, that (due to the rec keyword), the inputs to a package derivation may be other packages in the repository itself.

Rather than passing the inputs twice, we would prefer to pass those inputs from the repository automatically and allow for manually overriding defaults.

To achieve this, we will define a callPackage function with the following calling convention:

```
{
  lib1 = callPackage package1.nix { };
  program2 = callPackage package2.nix { someoverride = overriddenDerivation; };
}
```

We want callPackage to be a function of two arguments, with the following behavior:

- Import the given expression contained in the file of the first argument, and return a function. This function returns a package derivation that uses the inputs pattern.

- Determine the name of the arguments to the function (i.e., the names of the inputs to the package derivation).
- Pass default arguments from the repository set, and let us override those arguments if we wish to customize the package derivation.

Implementing `callPackage`

In this section, we will build up the `callPackages` pattern from scratch. To start, we need a way to obtain the argument names of a function (in this case, the function that takes "inputs" and produces a package derivation) at runtime. This is because we want to automatically pass such arguments.

Nix provides a builtin function to do this:

```
nix-repl> add = { a ? 3, b }: a+b
nix-repl> builtins.functionArgs add
{ a = true; b = false; }
```

In addition to returning the argument names, the attribute set returned by `functionArgs` indicates whether or not the argument has a default value. For our purposes, we are only interested in the argument names; we do not care about the default values right now.

The next step is to make `callPackage` automatically pass inputs to our package derivations based on the argument names we've just obtained with `functionArgs`.

To do this, we need two things:

- A package repository set containing package derivations that match the arguments names we've obtained
- A way to obtain an auto–populated attribute set combining the package repository and the return value of `functionArgs`.

The former is easy: we just have to set our package derivation's inputs to be package names in a repository, such as `nixpkgs`. For the latter, Nix provides another builtin function:

```
nix-repl> values = { a = 3; b = 5; c = 10; }
nix-repl> builtins.intersectAttrs values (builtins.functionArgs add)
{ a = true; b = false; }
nix-repl> builtins.intersectAttrs (builtins.functionArgs add) values
{ a = 3; b = 5; }
```

The `intersectAttrs` returns an attribute set whose names are the intersection of both arguments' attribute names, with the attribute values taken from the second argument.

This is all we need to do: we have obtained the argument names from a function, and populated these with an existing set of attributes. This is our simple implementation of `callPackage`:

```
nix-repl> callPackage = set: f: f (builtins.intersectAttrs (builtins.functionArgs f)
nix-repl> callPackage values add
8
nix-repl> with values; add { inherit a b; }
8
```

Let's dissect the above snippet:

- We define a `callPackage` variable which is a function.
- The first parameter to the `callPackage` function is a set of name–value pairs that may appear in the argument set of the function we wish to "autocall".
- The second parameter is the function to "autocall"
- We take the argument names of the function and intersect with the set of all values.
- Finally, we call the passed function `f` with the resulting intersection.

In the snippet above, we've also demonstrated that the `callPackage` call is equivalent to directly calling `add a b`.

We achieved most of what we wanted: to automatically call functions given a set of possible arguments. If an argument is not found within the set we used to call the function, then we receive an error (unless the function has variadic arguments denoted with `...`, as explained in the 5th pill).

The last missing piece is allowing users to override some of the parameters. We may not want to always call functions with values taken from the big set. Thus, we add a third parameter which takes a set of overrides:

```
nix-repl> callPackage = set: f: overrides: f ((builtins.intersectAttrs (builtins.fun
nix-repl> callPackage values add { }
8
nix-repl> callPackage values add { b = 12; }
15
```

Apart from the increasing number of parentheses, it should be clear that we simply take a set union between the default arguments and the overriding set.

Using callPackage to simplify the repository

Given our `callPackages`, we can simplify the repository expression in `default.nix`:

```
let
  nixpkgs = import <nixpkgs> { };
  allPkgs = nixpkgs // pkgs;
  callPackage =
    path: overrides:
    let
      f = import path;
    in
    f ((builtins.intersectAttrs (builtins.functionArgs f) allPkgs) // overrides);
  pkgs = with nixpkgs; {
    mkDerivation = import ./autotools.nix nixpkgs;
    hello = callPackage ./hello.nix { };
    graphviz = callPackage ./graphviz.nix { };
    graphvizCore = callPackage ./graphviz.nix { gdSupport = false; };
  };
in
pkgs
```

Let's examine this in detail:

- The expression above defines our own package repository, which we call `pkgs`, that contains `hello` along with our two variants of `graphviz`.

- In the `let` expression, we import `nixpkgs`. Note that previously, we referred to this import with the variable `pkgs`, but now that name is taken by the repository we are creating ourselves.
- We needed a way to pass `pkgs` to `callPackage` somehow. Instead of returning the set of packages directly from `default.nix`, we first assign it to a `let` variable and reuse it in `callPackage`.
- For convenience, in `callPackage` we first import the file instead of calling it directly. Otherwise we would have to write the `import` for each package.
- Since our expressions use packages from `nixpkgs`, in `callPackage` we use `allPkgs`, which is the union of `nixpkgs` and our packages.
- We moved `mkDerivation` into `pkgs` itself, so that it also gets passed automatically.

Note how easily we overrode arguments in the case of `graphviz` without `gd`. In addition, note how easy it was to merge two repositories: `nixpkgs` and our `pkgs`!

The reader should notice a magic thing happening. We're defining `pkgs` in terms of `callPackage`, and `callPackage` in terms of `pkgs`. That magic is possible thanks to lazy evaluation: `builtins.intersectAttrs` doesn't need to know the values in `allPkgs` in order to perform intersection, only the keys that do not require `callPackage` evaluation.

Conclusion

The "`callPackage`" pattern has simplified our repository considerably. We were able to import packages that require named arguments and call them automatically, given the set of all packages sourced from `nixpkgs`.

We've also introduced some useful builtin functions that allows us to introspect Nix functions and manipulate attributes. These builtin functions are not usually used when packaging software, but rather act as tools for packaging. They are documented in the Nix manual.

Writing a repository in Nix is an evolution of writing convenient functions for combining the packages. This pill demonstrates how Nix can be a generic tool to build and deploy software, and how suitable it is to create software repositories with our own conventions.

Next pill

In the next pill, we will talk about the "`override`" design pattern. The `graphvizCore` seems straightforward. It starts from `graphviz.nix` and builds it without `gd`. In the next pill, we will consider another point of view: starting from `pkgs.graphviz` and disabling `gd`?

Override Design Pattern

Welcome to the 14th Nix pill. In the previous 13th pill, we introduced the `callPackage` pattern and used it to simplify the composition of software in a repository.

The next design pattern is less necessary, but is useful in many cases and is a good exercise to learn more about Nix.

About composability

Functional languages are known for being able to compose functions. In particular, these languages gain expressivity from functions that manipulate an original value into a new value having the same structure. This allows us to compose multiple functions to perform the desired modifications.

In Nix, we mostly talk about **functions** that accept inputs in order to return **derivations**. In our world, we want utility functions that are able to manipulate those structures. These utilities add some useful properties to the original value, and we'd like to be able to apply more utilities on top of the result.

For example, let's say we have an initial derivation `drv` and we want to transform it into a `drv` with debugging information and custom patches:

```
debugVersion (applyPatches [ ./patch1.patch ./patch2.patch ] drv)
```

The final result should be the original derivation with some changes. This is both interesting and very different from other packaging approaches, which is a consequence of using a functional language to describe packages.

Designing such utilities is not trivial in a functional language without static typing, because understanding what can or cannot be composed is difficult. But we try to do our best.

The override pattern

In pill 12 we introduced the inputs design pattern. We do not return a derivation picking dependencies directly from the repository; rather we declare the inputs and let the callers pass the necessary arguments.

In our repository we have a set of attributes that import the expressions of the packages and pass these arguments, getting back a derivation. Let's take for example the `graphviz` attribute:

```
graphviz = import ./graphviz.nix { inherit mkDerivation gd fontconfig libjpeg bzip2;
```

If we wanted to produce a derivation of `graphviz` with a customized `gd` version, we would have to repeat most of the above plus specifying an alternative `gd`:

```
{
  mygraphviz = import ./graphviz.nix {
    inherit
      mkDerivation
      fontconfig
      libjpeg
      bzip2
      ;
    gd = customgd;
  };
}
```

That's hard to maintain. Using `callPackage` would be easier:

```
mygraphviz = callPackage ./graphviz.nix { gd = customgd; };
```

But we may still be diverging from the original graphviz in the repository.

We would like to avoid specifying the nix expression again. Instead, we would like to reuse the original `graphviz` attribute in the repository and add our overrides like so:

```
mygraphviz = graphviz.override { gd = customgd; };
```

The difference is obvious, as well as the advantages of this approach.

Note: that `.override` is not a "method" in the OO sense as you may think. Nix is a functional language. The=.override= is simply an attribute of a set.

The override implementation

Recall that the `graphviz` attribute in the repository is the derivation returned by the function imported from `graphviz.nix`. We would like to add a further attribute named "`override`" to the returned set.

Let's start by first creating a function "`makeOverridable`". This function will take two arguments: a function (that must return a set) and the set of original arguments to be passed to the function.

We will put this function in a `lib.nix`:

```
{
  makeOverridable =
    f: origArgs:
    let
      origRes = f origArgs;
    in
    origRes // { override = newArgs: f (origArgs // newArgs); };
}
```

`makeOverridable` takes a function and a set of original arguments. It returns the original returned set, plus a new `override` attribute.

This `override` attribute is a function taking a set of new arguments, and returns the result of the original function called with the original arguments unified with the new arguments. This is admittedly somewhat confusing, but the examples below should make it clear.

Let's try it with `nix repl`:

```
$ nix repl
nix-repl> :l lib.nix
Added 1 variables.
nix-repl> f = { a, b }: { result = a+b; }
nix-repl> f { a = 3; b = 5; }
{ result = 8; }
nix-repl> res = makeOverridable f { a = 3; b = 5; }
nix-repl> res
{ override = «lambda»; result = 8; }
nix-repl> res.override { a = 10; }
{ result = 15; }
```

Note that, as we specified above, the function `f` does not return the plain sum. Instead, it returns a set with the sum bound to the name `result`.

The variable `res` contains the result of the function call without any override. It's easy to see in the definition of `makeOverridable`. In addition, you can see that the new `override` attribute is a function.

Calling `res.override` with a set will invoke the original function with the overrides, as expected.

This is a good start, but we can't override again! This is because the returned set (with `result = 15`) does not have an `override` attribute of its own. This is bad; it breaks further composition.

The solution is simple: the `.override` function should make the result overridable again:

```
rec {
  makeOverridable =
    f: origArgs:
    let
      origRes = f origArgs;
    in
    origRes // { override = newArgs: makeOverridable f (origArgs // newArgs); };
}
```

Please note the `rec` keyword. It's necessary so that we can refer to `makeOverridable` from `makeOverridable` itself.

Now let's try overriding twice:

```
nix-repl> :l lib.nix
Added 1 variables.
nix-repl> f = { a, b }: { result = a+b; }
nix-repl> res = makeOverridable f { a = 3; b = 5; }
nix-repl> res2 = res.override { a = 10; }
nix-repl> res2
{ override = «lambda»; result = 15; }
nix-repl> res2.override { b = 20; }
{ override = «lambda»; result = 30; }
```

Success! The result is 30 (as expected) because `a` is overridden to 10 in the first override, and `b` is overridden to 20 in the second.

Now it would be nice if `callPackage` made our derivations overridable. This is an exercise for the reader.

Conclusion

The "`override`" pattern simplifies the way we customize packages starting from an existing set of packages. This opens a world of possibilities for using a central repository like `nixpkgs` and defining overrides on our local machine without modifying the original package.

We can dream of a custom, isolated `nix-shell` environment for testing `graphviz` with a custom `gd`:

```
debugVersion (graphviz.override { gd = customgd; })
```

Once a new version of the overridden package comes out in the repository, the customized package will make use of it automatically.

The key in Nix is to find powerful yet simple abstractions in order to let the user customize their environment with highest consistency and lowest maintenance time, by using predefined composable components.

Next pill

In the next pill, we will talk about Nix search paths. By "search path", we mean a place in the file system where Nix looks for expressions. This answers the question of where `<nixpkgs>` comes from.

Nix Search Paths

Welcome to the 15th Nix pill. In the previous 14th pill we have introduced the "override" pattern, useful for writing variants of derivations by passing different inputs.

Assuming you followed the previous posts, I hope you are now ready to understand nixpkgs. But we have to find nixpkgs in our system first! So this is the step: introducing some options and environment variables used by nix tools.

The NIX$_{PATH}$

The NIX$_{PATH}$ environment variable is very important. It's very similar to the PATH environment variable. The syntax is similar, several paths are separated by a colon :. Nix will then search for something in those paths from left to right.

Who uses NIX_PATH? The nix expressions! Yes, NIX_PATH is not of much use by the nix tools themselves, rather it's used when writing nix expressions.

In the shell for example, when you execute the command ping, it's being searched in the PATH directories. The first one found is the one being used.

In nix it's exactly the same, however the syntax is different. Instead of just typing ping you have to type <ping>. Yes, I know... you are already thinking of <nixpkgs>. However, don't stop reading here, let's keep going.

What's NIX_PATH good for? Nix expressions may refer to an "abstract" path such as <nixpkgs>, and it's possible to override it from the command line.

For ease we will use nix-instantiate --eval to do our tests. I remind you, nix–instantiate is used to evaluate nix expressions and generate the .drv files. Here we are not interested in building derivations, so evaluation is enough. It can be used for one–shot expressions.

Fake it a little

It's useless from a nix view point, but I think it's useful for your own understanding. Let's use PATH itself as NIX_PATH, and try to locate ping (or another binary if you don't have it).

```
$ nix-instantiate --eval -E '<ping>'
error: file `ping' was not found in the Nix search path (add it using $NIX_PATH or -
$ NIX_PATH=$PATH nix-instantiate --eval -E '<ping>'
/bin/ping
$ nix-instantiate -I /bin --eval -E '<ping>'
/bin/ping
```

Great. At first attempt nix obviously said could not be found anywhere in the search path. Note that the -I option accepts a single directory. Paths added with -I take precedence over NIX_PATH.

The NIX_PATH also accepts a different yet very handy syntax: "somename=somepath". That is, instead of searching inside a directory for a name, we specify exactly the value of that name.

```
$ NIX_PATH="ping=/bin/ping" nix-instantiate --eval -E '<ping>'
/bin/ping
$ NIX_PATH="ping=/bin/foo" nix-instantiate --eval -E '<ping>'
error: file `ping' was not found in the Nix search path (add it using $N
```

Note in the second case how Nix checks whether the path exists or not.

The path to repository

You are out of curiosity, right?

```
$ nix-instantiate --eval -E '<nixpkgs>'
/home/nix/.nix-defexpr/channels/nixpkgs
$ echo $NIX_PATH
nixpkgs=/home/nix/.nix-defexpr/channels/nixpkgs
```

You may have a different path, depending on how you added channels etc.. Anyway that's the whole point. The `<nixpkgs>` stranger that we used in our nix expressions, is referring to a path in the filesystem specified by `NIX_PATH`.

You can list that directory and realize it's simply a checkout of the nixpkgs repository at a specific commit (hint: `.version-suffix`).

The `NIX_PATH` variable is exported by `nix.sh`, and that's the reason why I always asked you to source nix.sh at the beginning of my posts.

You may wonder: then I can also specify a different nixpkgs path to, e.g., a `git checkout` of `nixpkgs`? Yes, you can and I encourage doing that. We'll talk about this in the next pill.

Let's define a path for our repository, then! Let's say all the `default.nix`, `graphviz.nix` etc. are under `/home/nix/mypkgs`:

```
$ export NIX_PATH=mypkgs=/home/nix/mypkgs:$NIX_PATH
$ nix-instantiate --eval '<mypkgs>'
{ graphviz = <code>; graphvizCore = <code>; hello = <code>; mkDerivation = <code>; }
```

Yes, `nix-build` also accepts paths with angular brackets. We first evaluate the whole repository (`default.nix`) and then pick the `graphviz` attribute.

A big word about nix-env

The nix-env command is a little different than `nix-instantiate` and `nix-build`. Whereas `nix-instantiate` and `nix-build` require a starting nix expression, `nix-env` does not.

You may be crippled by this concept at the beginning, you may think `nix-env` uses `NIX_PATH` to find the `nixpkgs` repository. But that's not it.

The `nix-env` command uses `~/.nix-defexpr`, which is also part of `NIX_PATH` by default, but that's only a coincidence. If you empty `NIX_PATH`, `nix-env` will still be able to find derivations because of `~/.nix-defexpr`.

So if you run `nix-env -i graphviz` inside your repository, it will install the nixpkgs one. Same if you set `NIX_PATH` to point to your repository.

In order to specify an alternative to `~/.nix-defexpr` it's possible to use the -f option:

```
$ nix-env -f '<mypkgs>' -i graphviz
warning: there are multiple derivations named `graphviz'; using the first one
replacing old `graphviz'
installing `graphviz'
```

Oh why did it say there's another derivation named `graphviz`? Because both `graphviz` and `graphvizCore` attributes in our repository have the name "graphviz" for the derivation:

```
$ nix-env -f '<mypkgs>' -qaP
graphviz      graphviz
graphvizCore  graphviz
hello         hello
```

By default `nix-env` parses all derivations and uses the derivation names to interpret the command line. So in this case "graphviz" matched two derivations. Alternatively, like for `nix-build`, one can use -A to specify an attribute name instead of a derivation name:

```
$ nix-env -f '<mypkgs>' -i -A graphviz
replacing old `graphviz'
installing `graphviz'
```

This form, other than being more precise, it's also faster because `nix-env` does not have to parse all the derivations.

For completeness: you must install `graphvizCore` with -A, since without the -A switch it's ambiguous.

In summary, it may happen when playing with nix that `nix-env` picks a different derivation than `nix-build`. In that case you probably specified `NIX_PATH`, but `nix-env` is instead looking into `~/.nix-defexpr`.

Why is `nix-env` having this different behavior? I don't know specifically by myself either, but the answers could be:

- `nix-env` tries to be generic, thus it does not look for `nixpkgs` in `NIX_PATH`, rather it looks in `~/.nix-defexpr`.
- `nix-env` is able to merge multiple trees in `~/.nix-defexpr` by looking at all the possible derivations

It may also happen to you **that you cannot match a derivation name when installing**, because of the derivation name vs -A switch described above. Maybe `nix-env` wanted to be more friendly in this case for default user setups.

It may or may not make sense for you, or it's like that for historical reasons, but that's how it works currently, unless somebody comes up with a better idea.

Conclusion

The `NIX_PATH` variable is the search path used by nix when using the angular brackets syntax. It's possible to refer to "abstract" paths inside nix expressions and define the "concrete" path by means of `NIX_PATH`, or the usual `-I` flag in nix tools.

We've also explained some of the uncommon `nix-env` behaviors for newcomers. The `nix-env` tool does not use `NIX_PATH` to search for packages, but rather for `~/.nix-defexpr`. Beware of that!

In general do not abuse `NIX_PATH`, when possible use relative paths when writing your own nix expressions. Of course, in the case of `<nixpkgs>` in our repository, that's a perfectly fine usage of `NIX_PATH`. Instead, inside our repository itself, refer to expressions with relative paths like `./hello.nix`.

Next pill

...we will finally dive into `nixpkgs`. Most of the techniques we have developed in this series are already in `nixpkgs`, like `mkDerivation`, `callPackage`, `override`, etc., but of course better. With time, those base utilities get enhanced by the community with more features in order to handle more and more use cases and in a more general way.

Nixpkgs Parameters

Welcome to the 16th Nix pill. In the previous 15th pill we've realized how nix finds expressions with the angular brackets syntax, so that we finally know where `<nixpkgs>` is located on our system.

We can start diving into the nixpkgs repository, through all the various tools and design patterns. Please note that also `nixpkgs` has its own manual, underlying the difference between the general `nix` language and the `nixpkgs` repository.

The default.nix expression

We will not start inspecting packages at the beginning, rather the general structure of `nixpkgs`.

In our custom repository we created a `default.nix` which composed the expressions of the various packages.

Also `nixpkgs` has its own default.nix, which is the one being loaded when referring to `<nixpkgs>`. It does a simple thing: check whether the `nix` version is at least 1.7 (at the time of writing this blog post). Then import pkgs/top−level/all−packages.nix. From now on, we will refer to this set of packages as **pkgs**.

The `all-packages.nix` is then the file that composes all the packages. Note the `pkgs/` subdirectory, while nixos is in the `nixos/` subdirectory.

The `all-packages.nix` is a bit contrived. First of all, it's a function. It accepts a couple of interesting parameters:

- `system`: defaults to the current system
- `config`: defaults to null
- others...

The **system** parameter, as per comment in the expression, it's the system for which the packages will be built. It allows for example to install i686 packages on amd64 machines.

The **config** parameter is a simple attribute set. Packages can read some of its values and change the behavior of some derivations.

The system parameter

You will find this parameter in many other .nix expressions (e.g. release expressions). The reason is that, given pkgs accepts a system parameter, then whenever you want to import pkgs you also want to pass through the value of system. E.g.:

`myrelease.nix`:

```
{ system ? builtins.currentSystem }:

let pkgs = import <nixpkgs> { inherit system; };
...
```

Why is it useful? With this parameter it's very easy to select a set of packages for a particular system. For example:

```
nix-build -A psmisc --argstr system i686-linux
```

This will build the psmisc derivation for i686–linux instead of x86$_{64}$–linux. This concept is very similar to multi–arch of Debian.

The setup for cross compiling is also in nixpkgs, however it's a little contrived to talk about it and I don't know much of it either.

The config parameter

I'm sure on the wiki or other manuals you've read about ~/.config/nixpkgs/config.nix (previously ~/.nixpkgs/config.nix) and I'm sure you've wondered whether that's hardcoded in nix. It's not, it's in nixpkgs.

The all–packages.nix expression accepts the config parameter. If it's null, then it reads the NIXPKGS_CONFIG environment variable. If not specified, nixpkgs will pick $HOME/.config/nixpkgs/config.nix.

After determining config.nix, it will be imported as a nix expression, and that will be the value of config (in case it hasn't been passed as parameter to import <nixpkgs>).

The config is available in the resulting repository:

```
$ nix repl
nix-repl> pkgs = import <nixpkgs> {}
nix-repl> pkgs.config
{ }
nix-repl> pkgs = import <nixpkgs> { config = { foo = "bar"; }; }
nix-repl> pkgs.config
{ foo = "bar"; }
```

What attributes go in config is a matter of convenience and conventions.

For example, config.allowUnfree is an attribute that forbids building packages that have an unfree license by default. The config.pulseaudio setting tells whether to build packages with pulseaudio support or not where applicable and when the derivation obeys to the setting.

About .nix functions

A .nix file contains a nix expression. Thus it can also be a function. I remind you that nix–build expects the expression to return a derivation. Therefore it's natural to return straight a derivation from a .nix file. However, it's also very natural for the .nix file to accept some parameters, in order to tweak the derivation being returned.

In this case, nix does a trick:

- If the expression is a derivation, build it.
- If the expression is a function, call it and build the resulting derivation.

For example you can nix–build the .nix file below:

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

pkgs.psmisc
```

Nix is able to call the function because the pkgs parameter has a default value. This allows you to pass a different value for pkgs using the --arg option.

Does it work if you have a function returning a function that returns a derivation? No, Nix only calls the function it encounters once.

Conclusion

We've unleashed the `<nixpkgs>` repository. It's a function that accepts some parameters, and returns the set of all packages. Due to laziness, only the accessed derivations will be built.

You can use this repository to build your own packages as we've seen in the previous pill when creating our own repository.

Lately I'm a little busy with the NixOS 14.11 release and other stuff, and I'm also looking toward migrating from blogger to a more coder–oriented blogging platform. So sorry for the delayed and shorter pills :)

Next pill

...we will talk about overriding packages in the `nixpkgs` repository. What if you want to change some options of a library and let all other packages pick the new library? One possibility is to use, like described above, the `config` parameter when applicable. The other possibility is to override derivations.

Nixpkgs Overriding Packages

Welcome to the 17th Nix pill. In the previous 16th pill we have started to dive into the nixpkgs repository. `Nixpkgs` is a function, and we've looked at some parameters like `system` and `config`.

Today we'll talk about a special attribute: `config.packageOverrides`. Overriding packages in a set with fixed point can be considered another design pattern in nixpkgs.

Overriding a package

Recall the override design pattern from the nix pill 14. Instead of calling a function with parameters directly, we make the call (function + parameters) overridable.

We put the override function in the returned attribute set of the original function call.

Take for example `graphviz`. It has an input parameter `xorg`. If it's null, then `graphviz` will build without X support.

```
$ nix repl
nix-repl> :l <nixpkgs>
Added 4360 variables.
nix-repl> :b graphviz.override { withXorg = false; }
```

This will build `graphviz` without X support, it's as simple as that.

However, let's say a package `P` depends on `graphviz`, how do we make `P` depend on the new `graphviz` without X support?

In an imperative world...

...you could do something like this:

```
pkgs = import <nixpkgs> {};
pkgs.graphviz = pkgs.graphviz.override { withXorg = false; };
build(pkgs.P)
```

Given `pkgs.P` depends on `pkgs.graphviz`, it's easy to build `P` with the replaced `graphviz`. In a pure functional language it's not that easy because you can assign to variables only once.

Fixed point

The fixed point with lazy evaluation is crippling but about necessary in a language like Nix. It lets us achieve something similar to what we'd do imperatively.

Follows the definition of fixed point in nixpkgs:

```
{
  # Take a function and evaluate it with its own returned value.
  fix =
    f:
    let
      result = f result;
    in
    result;
}
```

It's a function that accepts a function `f`, calls `f result` on the result just returned by `f result` and returns it. In other words it's `f(f(f(....`

At first sight, it's an infinite loop. With lazy evaluation it isn't, because the call is done only when needed.

```
nix-repl> fix = f: let result = f result; in result
nix-repl> pkgs = self: { a = 3; b = 4; c = self.a+self.b; }
nix-repl> fix pkgs
{ a = 3; b = 4; c = 7; }
```

Without the `rec` keyword, we were able to refer to `a` and `b` of the same set.

- First `pkgs` gets called with an unevaluated thunk `(pkgs(pkgs(...)`
- To set the value of `c` then `self.a` and `self.b` are evaluated.
- The `pkgs` function gets called again to get the value of `a` and `b`.

The trick is that `c` is not needed to be evaluated in the inner call, thus it doesn't go in an infinite loop.

Won't go further with the explanation here. A good post about fixed point and Nix can be found here.

Overriding a set with fixed point

Given that `self.a` and `self.b` refer to the passed set and not to the literal set in the function, we're able to override both `a` and `b` and get a new value for `c`:

```
nix-repl> overrides = { a = 1; b = 2; }
nix-repl> let newpkgs = pkgs (newpkgs // overrides); in newpkgs
{ a = 3; b = 4; c = 3; }
nix-repl> let newpkgs = pkgs (newpkgs // overrides); in newpkgs // overrides
{ a = 1; b = 2; c = 3; }
```

In the first case we computed pkgs with the overrides, in the second case we also included the overridden attributes in the result.

Overriding nixpkgs packages

We've seen how to override attributes in a set such that they get recursively picked by dependent attributes. This approach can be used for derivations too, after all `nixpkgs` is a giant set of attributes that depend on each other.

To do this, `nixpkgs` offers `config.packageOverrides`. So `nixpkgs` returns a fixed point of the package set, and `packageOverrides` is used to inject the overrides.

Create a `config.nix` file like this somewhere:

```
{
    packageOverrides = pkgs: {
    graphviz = pkgs.graphviz.override {
      # disable xorg support
      withXorg = false;
    };
  };
}
```

Now we can build e.g. `asciidoc-full` and it will automatically use the overridden `graphviz`:

```
nix-repl> pkgs = import <nixpkgs> { config = import ./config.nix; }
nix-repl> :b pkgs.asciidoc-full
```

Note how we pass the `config` with `packageOverrides` when importing `nixpkgs`. Then `pkgs.asciidoc-full` is a derivation that has `graphviz` input (`pkgs.asciidoc` is the lighter version and doesn't use `graphviz` at all).

Since there's no version of `asciidoc` with `graphviz` without X support in the binary cache, Nix will recompile the needed stuff for you.

The ~/.config/nixpkgs/config.nix file

In the previous pill we already talked about this file. The above `config.nix` that we just wrote could be the content of `~/.config/nixpkgs/config.nix` (or the deprecated location `~/.nixpkgs/config.nix`).

Instead of passing it explicitly whenever we import `nixpkgs`, it will be automatically imported by nixpkgs.

Conclusion

We've learned about a new design pattern: using fixed point for overriding packages in a package set.

Whereas in an imperative setting, like with other package managers, a library is installed replacing the old version and applications will use it, in Nix it's not that straight and simple. But it's more precise.

Nix applications will depend on specific versions of libraries, hence the reason why we have to recompile `asciidoc` to use the new `graphviz` library.

The newly built `asciidoc` will depend on the new `graphviz`, and old `asciidoc` will keep using the old `graphviz` undisturbed.

Next pill

...we will stop studying `nixpkgs` for a moment and talk about store paths. How does Nix compute the path in the store where to place the result of builds? How to add files to the store for which we have an integrity hash?

Nix Store Paths

Welcome to the 18th Nix pill. In the previous 17th pill we have scratched the surface of the `nixpkgs` repository structure. It is a set of packages, and it's possible to override such packages so that all other packages will use the overrides.

Before reading existing derivations, I'd like to talk about store paths and how they are computed. In particular we are interested in fixed store paths that depend on an integrity hash (e.g. a sha256), which is usually applied to source tarballs.

The way store paths are computed is a little contrived, mostly due to historical reasons. Our reference will be the Nix source code.

Source paths

Let's start simple. You know nix allows relative paths to be used, such that the file or directory is stored in the nix store, that is `./myfile` gets stored into `/nix/store/.......` We want to understand how is the store path generated for such a file:

```
$ echo mycontent > myfile
```

I remind you, the simplest derivation you can write has a `name`, a `builder` and the `system`:

```
$ nix repl
nix-repl> derivation { system = "x86_64-linux"; builder = ./myfile; name = "foo"; }
«derivation /nix/store/y4h73bmrc9ii5bxg6i7ck6hsf5gqv8ck-foo.drv»
```

Now inspect the .drv to see where is `./myfile` being stored:

```
$ nix derivation show /nix/store/y4h73bmrc9ii5bxg6i7ck6hsf5gqv8ck-foo.drv
{
  "/nix/store/y4h73bmrc9ii5bxg6i7ck6hsf5gqv8ck-foo.drv": {
    "outputs": {
      "out": {
        "path": "/nix/store/hs0yi5n5nw6micqhy8l1igkbhqdkzqa1-foo"
      }
    },
    "inputSrcs": [
      "/nix/store/xv2iccirbrvklck36f1g7vldn5v58vck-myfile"
    ],
    "inputDrvs": {},
    "platform": "x86_64-linux",
    "builder": "/nix/store/xv2iccirbrvklck36f1g7vldn5v58vck-myfile",
    "args": [],
    "env": {
      "builder": "/nix/store/xv2iccirbrvklck36f1g7vldn5v58vck-myfile",
      "name": "foo",
      "out": "/nix/store/hs0yi5n5nw6micqhy8l1igkbhqdkzqa1-foo",
      "system": "x86_64-linux"
    }
  }
}
```

Great, how did nix decide to use `xv2iccirbrvklck36f1g7vldn5v58vck` ? Keep looking at the nix comments.

Note: doing `nix-store --add myfile` will store the file in the same store path.

Step 1, compute the hash of the file

The comments tell us to first compute the sha256 of the NAR serialization of the file. Can be done in two ways:

```
$ nix-hash --type sha256 myfile
2bfef67de873c54551d884fdab3055d84d573e654efa79db3c0d7b98883f9ee3
```

Or:

```
$ nix-store --dump myfile|sha256sum
2bfef67de873c54551d884fdab3055d84d573e654efa79db3c0d7b98883f9ee3
```

In general, Nix understands two contents: flat for regular files, or recursive for NAR serializations which can be anything.

Step 2, build the string description

Then nix uses a special string which includes the hash, the path type and the file name. We store this in another file:

```
$ echo -n "source:sha256:2bfef67de873c54551d884fdab3055d84d573e654efa79db3c0d7b98883
```

Step 3, compute the final hash

Finally the comments tell us to compute the base–32 representation of the first 160 bits (truncation) of a sha256 of the above string:

```
$ nix-hash --type sha256 --truncate --base32 --flat myfile.str
xv2iccirbrvklck36f1g7vldn5v58vck
```

Output paths

Output paths are usually generated for derivations. We use the above example because it's simple. Even if we didn't build the derivation, nix knows the out path `hs0yi5n5nw6micqhy8l1igkbhqdkzqa1`. This is because the out path only depends on inputs.

It's computed in a similar way to source paths, except that the .drv is hashed and the type of derivation is `output:out`. In case of multiple outputs, we may have different `output:<id>`.

At the time nix computes the out path, the .drv contains an empty string for each out path. So what we do is getting our .drv and replacing the out path with an empty string:

```
$ cp -f /nix/store/y4h73bmrc9ii5bxg6i7ck6hsf5gqv8ck-foo.drv myout.drv
$ sed -i 's,/nix/store/hs0yi5n5nw6micqhy8l1igkbhqdkzqa1-foo,,g' myout.drv
```

The `myout.drv` is the .drv state in which nix is when computing the out path for our derivation:

```
$ sha256sum myout.drv
1bdc41b9649a0d59f270a92d69ce6b5af0bc82b46cb9d9441ebc6620665f40b5  myout.drv
$ echo -n "output:out:sha256:1bdc41b9649a0d59f270a92d69ce6b5af0bc82b46cb9d9441ebc662
$ nix-hash --type sha256 --truncate --base32 --flat myout.str
hs0yi5n5nw6micqhy8l1igkbhqdkzqa1
```

Then nix puts that out path in the .drv, and that's it.

In case the .drv has input derivations, that is it references other .drv, then such .drv paths are replaced by this same algorithm which returns a hash.

In other words, you get a final .drv where every other .drv path is replaced by its hash.

Fixed–output paths

Finally, the other most used kind of path is when we know beforehand an integrity hash of a file. This is usual for tarballs.

A derivation can take three special attributes: `outputHashMode`, `outputHash` and `outputHashAlgo` which are well documented in the nix manual.

The builder must create the out path and make sure its hash is the same as the one declared with `outputHash`.

Let's say our builder should create a file whose contents is `mycontent`:

```
$ echo mycontent > myfile
$ sha256sum myfile
f3f3c4763037e059b4d834eaf68595bbc02ba19f6d2a500dce06d124e2cd99bb  myfile
nix-repl> derivation { name = "bar"; system = "x86_64-linux"; builder = "none"; outp
«derivation /nix/store/ymsf5zcqr9wlkkqdjwhqllgwa97rff5i-bar.drv»
```

Inspect the .drv and see that it also stored the fact that it's a fixed–output derivation with sha256 algorithm, compared to the previous examples:

```
$ nix derivation show /nix/store/ymsf5zcqr9wlkkqdjwhqllgwa97rff5i-bar.drv
{
  "/nix/store/ymsf5zcqr9wlkkqdjwhqllgwa97rff5i-bar.drv": {
    "outputs": {
      "out": {
        "path": "/nix/store/a00d5f71k0vp5a6klkls0mvr1f7sx6ch-bar",
        "hashAlgo": "sha256",
        "hash": "f3f3c4763037e059b4d834eaf68595bbc02ba19f6d2a500dce06d124e2cd99bb"
      }
    },
[...]
}
```

It doesn't matter which input derivations are being used, the final out path must only depend on the declared hash.

What nix does is to create an intermediate string representation of the fixed–output content:

```
$ echo -n "fixed:out:sha256:f3f3c4763037e059b4d834eaf68595bbc02ba19f6d2a500dce06d124
$ sha256sum mycontent.str
423e6fdef56d53251c5939359c375bf21ea07aaa8d89ca5798fb374dbcfd7639  myfile.str
```

Then proceed as it was a normal derivation output path:

```
$ echo -n "output:out:sha256:423e6fdef56d53251c5939359c375bf21ea07aaa8d89ca5798fb374
$ nix-hash --type sha256 --truncate --base32 --flat myfile.str
a00d5f71k0vp5a6klkls0mvr1f7sx6ch
```

Hence, the store path only depends on the declared fixed–output hash.

Conclusion

There are other types of store paths, but you get the idea. Nix first hashes the contents, then creates a string description, and the final store path is the hash of this string.

Also we've introduced some fundamentals, in particular the fact that Nix knows beforehand the out path of a derivation since it only depends on the inputs. We've also introduced fixed-output derivations which are especially used by the nixpkgs repository for downloading and verifying source tarballs.

Next pill

...we will introduce `stdenv`. In the previous pills we rolled our own `mkDerivation` convenience function for wrapping the builtin derivation, but the `nixpkgs` repository also has its own convenience functions for dealing with `autotools` projects and other build systems.

Fundamentals of Stdenv

Welcome to the 19th Nix pill. In the previous 18th pill we dived into the algorithm used by Nix to compute the store paths, and also introduced fixed-output store paths.

This time we will instead look into `nixpkgs`, in particular one of its core derivations: `stdenv`.

The `stdenv` is not treated as a special derivation by Nix, but it's very important for the `nixpkgs` repository. It serves as a base for packaging software. It is used to pull in dependencies such as the GCC toolchain, GNU make, core utilities, patch and diff utilities, and so on: basic tools needed to compile a huge pile of software currently present in `nixpkgs`.

What is stdenv?

First of all, `stdenv` is a derivation, and it's a very simple one:

```
$ nix-build '<nixpkgs>' -A stdenv
/nix/store/k4jklkcag4zq4xkqhkpy156mgfm34ipn-stdenv
$ ls -R result/
result/:
nix-support/  setup

result/nix-support:
propagated-user-env-packages
```

It has just two files: `/setup` and `/nix-support/propagated-user-env-packages`. Don't worry about the latter. It's empty, in fact. The important file is `/setup`.

How can this simple derivation pull in all of the toolchain and basic tools needed to compile packages? Let's look at the runtime dependencies:

```
$ nix-store -q --references result
/nix/store/3a45nb37s0ndljp68228snsqr3qsyp96-bzip2-1.0.6
/nix/store/a457ywa1haa0sgr9g7a1pgldrg3s798d-coreutils-8.24
/nix/store/zmd4jk4db5lgxb8l93mhkvr3x92g2sx2-bash-4.3-p39
/nix/store/47sfpm2qclpqvrzijizimk4md1739b1b-gcc-wrapper-4.9.3
...
```

How can it be? The package must be referring to those other packages somehow. In fact, they are hardcoded in the /setup file:

```
$ head result/setup
export SHELL=/nix/store/zmd4jk4db5lgxb8l93mhkvr3x92g2sx2-bash-4.3-p39/bin/bash
initialPath="/nix/store/a457ywa1haa0sgr9g7a1pgldrg3s798d-coreutils-8.24 ..."
defaultNativeBuildInputs="/nix/store/sgwq15xg00xnm435gjicspm048rqg9y6-patchelf-0.8 .
```

The setup file

Remember our generic builder.sh in Pill 8? It sets up a basic PATH, unpacks the source and runs the usual autotools commands for us.

The stdenv setup file is exactly that. It sets up several environment variables like PATH and creates some helper bash functions to build a package. I invite you to read it.

The hardcoded toolchain and utilities are used to initially fill up the environment variables so that it's more pleasant to run common commands, similar to what we did with our builder with baseInputs and buildInputs.

The build with stdenv works in phases. Phases are like unpackPhase, configurePhase, buildPhase, checkPhase, installPhase, fixupPhase. You can see the default list in the genericBuild function.

What genericBuild does is just run these phases. Default phases are just bash functions. You can easily read them.

Every phase has hooks to run commands before and after the phase has been executed. Phases can be overwritten, reordered, whatever, it's just bash code.

How to use this file? Like our old builder. To test it, we enter a fake empty derivation, source the stdenv setup, unpack the hello sources and build it:

```
$ nix-shell -E 'derivation { name = "fake"; builder = "fake"; system = "x86_64-linux
nix-shell$ unset PATH
nix-shell$ source /nix/store/k4jklkcag4zq4xkqhkpy156mgfm34ipn-stdenv/setup
nix-shell$ tar -xf hello-2.10.tar.gz
nix-shell$ cd hello-2.10
nix-shell$ configurePhase
...
nix-shell$ buildPhase
...
```

/I unset PATH to further show that the stdenv is sufficiently self–contained to build autotools packages that have no other dependencies./

So we ran the configurePhase function and buildPhase function and they worked. These bash functions should be self–explanatory. You can read the code in the setup file.

How the setup file is built

Until now we worked with plain bash scripts. What about the Nix side? The nixpkgs repository offers a useful function, like we did with our old builder. It is a wrapper around the raw derivation function which pulls in the stdenv for us, and runs genericBuild. It's stdenv.mkDerivation.

Note how stdenv is a derivation but it's also an attribute set which contains some other attributes, like mkDerivation. Nothing fancy here, just convenience.

Let's write a `hello.nix` expression using this newly discovered `stdenv`:

```
with import <nixpkgs> { };
stdenv.mkDerivation {
  name = "hello";
  src = ./hello-2.10.tar.gz;
}
```

Don't be scared by the `with` expression. It pulls the `nixpkgs` repository into scope, so we can directly use `stdenv`. It looks very similar to the hello expression in Pill 8.

It builds, and runs fine:

```
$ nix-build hello.nix
...
/nix/store/6y0mzdarm5qxfafvn2zm9nr01d1j0a72-hello
$ result/bin/hello
Hello, world!
```

The stdenv.mkDerivation builder

Let's take a look at the builder used by `mkDerivation`. You can read the code here in nixpkgs:

```
{
  # ...
  builder = attrs.realBuilder or shell;
  args =
    attrs.args or [
      "-e"
      (attrs.builder or ./default-builder.sh)
    ];
  stdenv = result;
  # ...
}
```

Also take a look at our old derivation wrapper in previous pills! The builder is bash (that shell variable), the argument to the builder (bash) is `default-builder.sh`, and then we add the environment variable `$stdenv` in the derivation which is the `stdenv` derivation.

You can open default–builder.sh and see what it does:

```
source $stdenv/setup
genericBuild
```

It's what we did in Pill 10 to make the derivations `nix-shell` friendly. When entering the shell, the setup file only sets up the environment without building anything. When doing `nix-build`, it actually runs the build process.

To get a clear understanding of the environment variables, look at the .drv of the hello derivation:

```
$ nix derivation show $(nix-instantiate hello.nix)
warning: you did not specify '--add-root'; the result might be removed by the garbag
{
  "/nix/store/abwj50lycl0m515yblnrvwyydlhhqvj2-hello.drv": {
    "outputs": {
      "out": {
```

```
        "path": "/nix/store/6y0mzdarm5qxfafvn2zm9nr01d1j0a72-hello"
      }
    },
    "inputSrcs": [
      "/nix/store/9krlzvny65gdc8s7kpb6lkx8cd02c25b-default-builder.sh",
      "/nix/store/svc70mmzrlgq42m9acs0prsmci7ksh6h-hello-2.10.tar.gz"
    ],
    "inputDrvs": {
      "/nix/store/hcgwbx42mcxr7ksnv0i1fg7kw6jvxshb-bash-4.4-p19.drv": [
        "out"
      ],
      "/nix/store/sfxh3ybqh97cgl4s59nrpi78kgcc8f3d-stdenv-linux.drv": [
        "out"
      ]
    },
    "platform": "x86_64-linux",
    "builder": "/nix/store/q1g0rl8zfmz7r371fp5p42p4acmv297d-bash-4.4-p19/bin/bash",
    "args": [
      "-e",
      "/nix/store/9krlzvny65gdc8s7kpb6lkx8cd02c25b-default-builder.sh"
    ],
    "env": {
      "buildInputs": "",
      "builder": "/nix/store/q1g0rl8zfmz7r371fp5p42p4acmv297d-bash-4.4-p19/bin/bash"
      "configureFlags": "",
      "depsBuildBuild": "",
      "depsBuildBuildPropagated": "",
      "depsBuildTarget": "",
      "depsBuildTargetPropagated": "",
      "depsHostBuild": "",
      "depsHostBuildPropagated": "",
      "depsTargetTarget": "",
      "depsTargetTargetPropagated": "",
      "name": "hello",
      "nativeBuildInputs": "",
      "out": "/nix/store/6y0mzdarm5qxfafvn2zm9nr01d1j0a72-hello",
      "propagatedBuildInputs": "",
      "propagatedNativeBuildInputs": "",
      "src": "/nix/store/svc70mmzrlgq42m9acs0prsmci7ksh6h-hello-2.10.tar.gz",
      "stdenv": "/nix/store/6kz2vbh98s2r1pfshidkzhiy2s2qdw0a-stdenv-linux",
      "system": "x86_64-linux"
    }
  }
}
```

It's so short I decided to paste it entirely above. The builder is bash, with `-e default-builder.sh` arguments. Then you can see the `src` and `stdenv` environment variables.

The last bit, the `unpackPhase` in the setup, is used to unpack the sources and enter the directory. Again, like we did in our old builder.

Conclusion

The `stdenv` is the core of the `nixpkgs` repository. All packages use the `stdenv.mkDerivation` wrapper instead of the raw derivation. It does a bunch of operations for us and also sets up a pleasant build environment.

The overall process is simple:

- `nix-build`
- `bash -e default-builder.sh`
- `source $stdenv/setup`
- `genericBuild`

That's it. Everything you need to know about the stdenv phases is in the setup file.

Really, take your time to read that file. Don't forget that juicy docs are also available in the nixpkgs manual.

Next pill...

...we will talk about how to add dependencies to our packages with `buildInputs` and `propagatedBuildInputs`, and influence downstream builds with setup hooks and env hooks. These concepts are crucial to how `nixpkgs` packages are composed.

Basic Dependencies and Hooks

Welcome to the 20th Nix pill. In the previous 19th pill we introduced Nixpkgs' stdenv, including `setup.sh` script, `default-builder.sh` helper script, and `stdenv.mkDerivation` builder. We focused on how stdenv is put together, and how it's used, and a bit about the phases of `genericBuild`.

This time, we'll focus on the interaction of packages built with `stdenv.mkDerivation`. Packages need to depend on each other, of course. For this we have `buildInputs` and `propagatedBuildInputs` attributes. We've also found that dependencies sometimes need to influence their dependents in ways the dependents can't or shouldn't predict. For this we have setup hooks and env hooks. Together, these 4 concepts support almost all build–time package interactions.

Note: The complexity of the dependencies and hooks infrastructure has increased, over time, to support cross compilation. Once you learn the core concepts, you will be able to understand the extra complexity. As a starting point, you might want to refer to nixpkgs commit 6675f0a5, the last version of stdenv without cross–compilation complexity.

The `buildInputs` Attribute

For the simplest dependencies where the current package directly needs another, we use the `buildInputs` attribute. This is exactly the pattern used in our builder in Pill 8. To demo this, let's build GNU Hello, and then another package which provides a shell script that =exec=s it.

```
let

  nixpkgs = import <nixpkgs> { };

  inherit (nixpkgs) stdenv fetchurl which;
```

```
  actualHello = stdenv.mkDerivation {
    name = "hello-2.3";

    src = fetchurl {
      url = "mirror://gnu/hello/hello-2.3.tar.bz2";
      sha256 = "0c7vijq8y68bpr7g6dh1gny0bff8qq81vnp4ch8pjzvg56wb3js1";
    };
  };

  wrappedHello = stdenv.mkDerivation {
    name = "hello-wrapper";

    buildInputs = [
      actualHello
      which
    ];

    unpackPhase = "true";

    installPhase = ''
      mkdir -p "$out/bin"
      echo "#! ${stdenv.shell}" >> "$out/bin/hello"
      echo "exec $(which hello)" >> "$out/bin/hello"
      chmod 0755 "$out/bin/hello"
    '';
  };
in
wrappedHello
```

Notice that the wrappedHello derivation finds the `hello` binary from the `PATH`. This works because stdenv contains something like:

```
pkgs=""
for i in $buildInputs; do
    findInputs $i
done
```

where `findInputs` is defined like:

```
findInputs() {
    local pkg=$1

    ## Don't need to repeat already processed package
    case $pkgs in
        *\ $pkg\ *)
            return 0
            ;;
    esac

    pkgs="$pkgs $pkg "

    ## More goes here in reality that we can ignore for now.
}
```

then after this is run:

```
for i in $pkgs; do
    addToEnv $i
done
```

where addToEnv is defined like:

```
addToEnv() {
    local pkg=$1

    if test -d $1/bin; then
        addToSearchPath _PATH $1/bin
    fi

    ## More goes here in reality that we can ignore for now.
}
```

The addToSearchPath call adds $1/bin to _PATH if the former exists (code here). Once all the packages in buildInputs have been processed, then content of _PATH is added to PATH, as follows:

```
PATH="${_PATH-}${_PATH:+${PATH:+:}}$PATH"
```

With the real hello on the PATH, the installPhase should hopefully make sense.

The **propagatedBuildInputs** Attribute

The buildInputs covers direct dependencies, but what about indirect dependencies where one package needs a second package which needs a third? Nix itself handles this just fine, understanding various dependency closures as covered in previous builds. But what about the conveniences that buildInputs provides, namely accumulating in pkgs environment variable and inclusion of «pkg»/bin directories on the PATH? For this, stdenv provides the propagatedBuildInputs:

```
let

  nixpkgs = import <nixpkgs> { };

  inherit (nixpkgs) stdenv fetchurl which;

  actualHello = stdenv.mkDerivation {
    name = "hello-2.3";

    src = fetchurl {
      url = "mirror://gnu/hello/hello-2.3.tar.bz2";
      sha256 = "0c7vijq8y68bpr7g6dh1gny0bff8qq81vnp4ch8pjzvg56wb3js1";
    };
  };

  intermediary = stdenv.mkDerivation {
    name = "middle-man";

    propagatedBuildInputs = [ actualHello ];

    unpackPhase = "true";
```

```
    installPhase = ''
      mkdir -p "$out"
    '';
  };

  wrappedHello = stdenv.mkDerivation {
    name = "hello-wrapper";

    buildInputs = [
      intermediary
      which
    ];

    unpackPhase = "true";

    installPhase = ''
      mkdir -p "$out/bin"
      echo "#! ${stdenv.shell}" >> "$out/bin/hello"
      echo "exec $(which hello)" >> "$out/bin/hello"
      chmod 0755 "$out/bin/hello"
    '';
  };
in
wrappedHello
```

See how the intermediate package has a `propagatedBuildInputs` dependency, but the wrapper only needs a `buildInputs` dependency on the intermediary.

How does this work? You might think we do something in Nix, but actually it's done not at eval time but at build time in bash. let's look at part of the `fixupPhase` of stdenv:

```
fixupPhase() {

    ## Elided

    if test -n "$propagatedBuildInputs"; then
        mkdir -p "$out/nix-support"
        echo "$propagatedBuildInputs" > "$out/nix-support/propagated-build-inputs"
    fi

    ## Elided

}
```

This dumps the propagated build inputs in a so-named file in `$out/nix-support/`. Then, back in `findInputs` look at the lines at the bottom we elided before:

```
findInputs() {
    local pkg=$1

    ## More goes here in reality that we can ignore for now.

    if test -f $pkg/nix-support/propagated-build-inputs; then
        for i in $(cat $pkg/nix-support/propagated-build-inputs); do
```

```
            findInputs $i
        done
    fi
}
```

See how `findInputs` is actually recursive, looking at the propagated build inputs of each dependency, and those dependencies' propagated build inputs, etc.

We actually simplified the `findInputs` call site from before; `propagatedBuildInputs` is also looped over in reality:

```
pkgs=""
for i in $buildInputs $propagatedBuildInputs; do
    findInputs $i
done
```

This demonstrates an important point. For the *current* package alone, it doesn't matter whether a dependency is propagated or not. It will be processed the same way: called with `findInputs` and `addToEnv`. (The packages discovered by `findInputs`, which are also accumulated in `pkgs` and passed to `addToEnv`, are also the same in both cases.) Downstream however, it certainly does matter because only the propagated immediate dependencies are put in the `$out/nix-support/propagated-build-inputs`.

Setup Hooks

As we mentioned above, sometimes dependencies need to influence the packages that use them in ways other than just *being* a dependency. [1] `propagatedBuildInputs` can actually be seen as an example of this: packages using that are effectively "injecting" those dependencies as extra `buildInputs` in their downstream dependents. But in general, a dependency might affect the packages it depends on in arbitrary ways. *Arbitrary* is the key word here. We could teach `setup.sh` things about upstream packages like `«pkg»/nix-support/propagated-build-inputs`, but not arbitrary interactions.

Setup hooks are the basic building block we have for this. In nixpkgs, a "hook" is basically a bash callback, and a setup hook is no exception. Let's look at the last part of `findInputs` we haven't covered:

```
findInputs() {
    local pkg=$1

    ## More goes here in reality that we can ignore for now.

    if test -f $pkg/nix-support/setup-hook; then
        source $pkg/nix-support/setup-hook
    fi

    ## More goes here in reality that we can ignore for now.

}
```

If a package includes the path `«pkg»/nix-support/setup-hook`, it will be sourced by any stdenv-based build including that as a dependency.

This is strictly more general than any of the other mechanisms introduced in this chapter. For example, try writing a setup hook that has the same effect as a *propagatedBuildInputs* entry. One can almost think of this as an escape hatch around Nix's normal isolation guarantees, and the principle that dependencies are immutable and inert. We're not actually doing something unsafe or modifying dependencies, but we are allowing arbitrary ad–hoc behavior. For this reason, setup–hooks should only be used as a last resort.

Environment Hooks

As a final convenience, we have environment hooks. Recall in Pill 12 how we created `NIX_CFLAGS_COMPILE` for `-I` flags and `NIX_LDFLAGS` for `-L` flags, in a similar manner to how we prepared the `PATH`. One point of ugliness was how anti–modular this was. It makes sense to build the `PATH` in a generic builder, because the `PATH` is used by the shell, and the generic builder is intrinsically tied to the shell. But `-I` and `-L` flags are only relevant to the C compiler. The stdenv isn't wedded to including a C compiler (though it does by default), and there are other compilers too which may take completely different flags.

As a first step, we can move that logic to a setup hook on the C compiler; indeed that's just what we do in CC Wrapper. [2] But this pattern comes up fairly often, so somebody decided to add some helper support to reduce boilerplate.

The other half of `addToEnv` is:

```
addToEnv() {
    local pkg=$1

    ## More goes here in reality that we can ignore for now.

    # Run the package-specific hooks set by the setup-hook scripts.
    for i in "${envHooks[@]}"; do
        $i $pkg
    done
}
```

Functions listed in `envHooks` are applied to every package passed to `addToEnv`. One can write a setup hook like:

```
anEnvHook() {
    local pkg=$1

    echo "I'm depending on \"$pkg\""
}

envHooks+=(anEnvHook)
```

and if one dependency has that setup hook then all of them will be so =echo=ed. Allowing dependencies to learn about their *sibling* dependencies is exactly what compilers need.

Next pill...

...I'm not sure! We could talk about the additional dependency types and hooks which cross compilation necessitates, building on our knowledge here to cover stdenv as it works today. We could talk about how nixpkgs is bootstrapped. Or we could talk about how `localSystem` and `crossSystem` are elaborated into the `buildPlatform`, `hostPlatform`, and `targetPlatform` each bootstrapping stage receives. Let us know which most interests you!

[1] We can now be precise and consider what `addToEnv` does alone the minimal treatment of a dependency: i.e. a package that is *just* a dependency would *only* have `addToEnv`

applied to it.

[2] It was called GCC Wrapper in the version of nixpkgs suggested for following along in this pill; Darwin and Clang support hadn't yet motivated the rename.