A conceptual introduction to Nix

Krzysztof Gogolewski

This talk

Nix explained.

This is not a tutorial on how to accomplish things with nix.

Why Nix?

Adds structure to your workflow.

nix repl: ghci

```
nix-repl> "a" + "b"
"ab"
nix-repl> map (x: x * 10) [ 2 3 ]
[ 20 30 ]
nix-repl> { a = true; b = 10; }.a
true
nix-repl> builtins.currentSystem
"x86_64-linux"
```

Something more interesting

Imperative languages:

Something more interesting

Imperative languages: side effect.

Something more interesting

Imperative languages: side effect.

Haskell: value of type `IO a`.

Something more interesting

Imperative languages: side effect.

Haskell: value of type `IO a`.

Nix: a derivation.

Derivation example

```
derivation {
  builder = "/bin/sh";
  args = ["-c" "echo hello > $out"];

  name = "write-hello";
  system = builtins.currentSystem;
}
```

Derivation example

```
nix-repl> derivation {
  builder = "/bin/sh";
  args = ["-c" "echo hello > $out"];

  name = "write-hello";
  system = builtins.currentSystem;
}
```

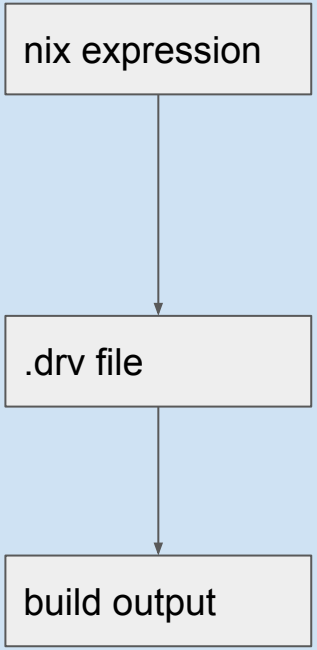
Derivation example: instantiation

```
nix-repl> derivation {
  builder = "/bin/sh";
  args = ["-c" "echo hello > $out"];

  name = "write-hello";
  system = builtins.currentSystem;
}
«derivation /nix/store/sbvd...s5kyz7pj9j05zx-write-hello.drv»
```

Derivation example: building

```
nix-repl> :b derivation {
  builder = "/bin/sh";
  args = ["-c" "echo hello > $out"];

  name = "write-hello";
  system = builtins.currentSystem;
}
this derivation produced the following outputs:

  out -> /nix/store/zs7nnwyzpy4...05qgw48v3k7xm5-write-hello
```

Derivation example

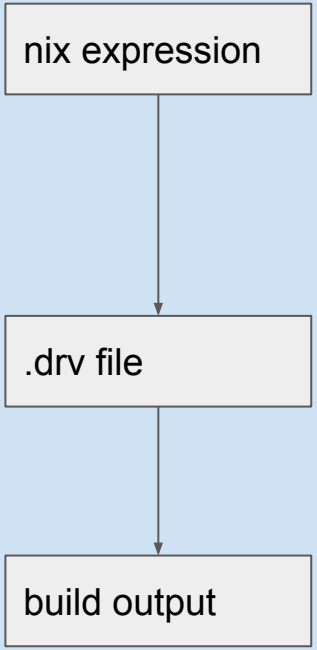



```
derivation { builder = "/bin/sh";
             args = ["-c" "echo hello > $out"];
             name = "write-hello";
             system = builtins.currentSystem;
           }
```

```
Derive([("out","/nix/store/zs7nnwyzpy46b4677805qgw48v3k7xm5-write-hello","","")],[],[],"x86_64-linux","/bin/sh",["-c","echo hello > $out"],[("builder","/bin/sh"),("name","write-hello"),("out","/nix/store/zs7nnwyzpy46b4677805qgw48v3k7xm5-write-hello"),("system","x86_64-linux")])
```

```
hello
```

Derivation example




```
d = derivation { builder = "/bin/sh";
                 args = ["-c" "echo hello > $out"];
                 name = "write-hello";
                 system = builtins.currentSystem;
               }
```

```
d.drvPath -> "/nix/store/sbvd...s5kyz7pj9j05zx-write-hello.drv"
```

```
"${d}"    -> "/nix/store/zs7n...4677805qgw48v3k7xm5-write-hello"
```

The output path

```
d = derivation { builder = "/bin/sh";
                 args = ["-c" "echo hello > $out"];
                 name = "write-hello";
                 system = builtins.currentSystem;
               }

"${d}"
```

The output path

```
d = derivation { builder = "/bin/sh";
                 args = ["-c" "echo hello >$out"];
                 name = "write-hello";
                 system = builtins.currentSystem;
               }

"${d}"
```

The output path

```
d = derivation { builder = "/bin/sh";
                 args = ["-c" "echo hello >${d}"];
                 name = "write-hello";
                 system = builtins.currentSystem;
               }

"${d}"
```

The output path

Infinite recursion

```
d = derivation { builder = "/bin/sh";
                 args = ["-c" "echo hello >${d}"];
                 name = "write-hello";
                 system = builtins.currentSystem;
               }

"${d}"
```

External script

```
derivation { builder = "/bin/sh";
             args = ["-c" "echo hello > $out"];
             name = "write-hello";
             system = builtins.currentSystem;
           }
```

External script

Create a file builder.sh:

```
echo "hello" > $out
```

```
derivation { builder = "/bin/sh";
             args = ["${./builder.sh}"];
             name = "write-hello";
             system = builtins.currentSystem;
           }
```

External script

Create a file builder.sh:

```
echo "hello" > $out
```

```
derivation { builder = "/bin/sh";
             args = ["${./builder.sh}"];
             name = "write-hello";
             system = builtins.currentSystem;
           }
```

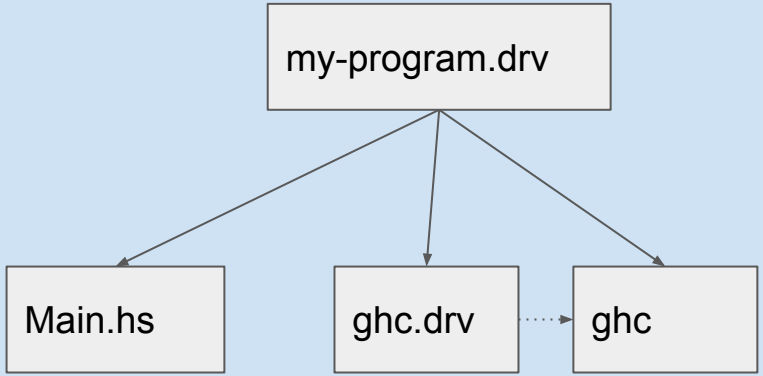
```
nix-repl> "${./builder.sh}"

"/nix/store/iidz0av5c24qhx3m4a1h9pzl25xwf03s-builder.sh"
```

Splicing

Splicing a **derivation** returns the path to the build output.

Splicing a **path** copies the file to the Nix store and returns its location.

```
let ghc = derivation ...; in
derivation {
  builder = "${ghc}/bin/ghc";
  args = ["${./Main.hs}"];
  name = "my-program";
  system = builtins.currentSystem;
}
```



Splicing

Splicing a **derivation** returns the path to the build output.

Splicing a **path** copies the file to the Nix store and returns its location.

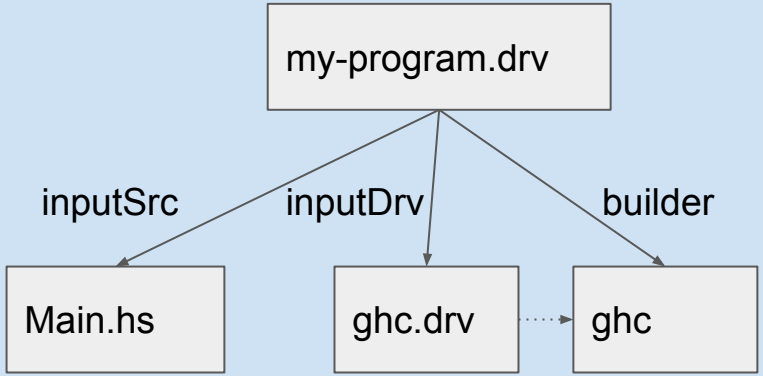
```
derivation {
  builder = "/nix/store/m26...ig5dm2z6z-ghc-8.10.4/bin/ghc";
  args = ["/nix/store/ayx...h2sf46x252vf5rrv9c-Main.hs"];
  name = "my-program";
  system = builtins.currentSystem;
}
```

Splicing

Splicing a **derivation** returns the path to the build output.

Splicing a **path** copies the file to the Nix store and returns its location.

```
let ghc = derivation ...; in
derivation {
  builder = "${ghc}/bin/ghc";
  args = ["${./Main.hs}"];
  name = "my-program";
  system = builtins.currentSystem;
}
```



Sandboxing

Derivations are built in a sandbox.

You can access only `/bin/sh`, files that were declared using a path `"${./path}"` or a derivation `"${d}"`

"The name Nix is derived from the Dutch word niks, meaning nothing; build actions do not see anything that has not been explicitly declared as an input."

-- Nix: A Safe and Policy-Free System for Software Deployment (2004)

Derivations as Haskell values

```
type Path = String
data Derivation = Derivation {
                    builder :: Path,
                    args :: [String],
                    inputSrcs :: [Path],
                    inputDrvs :: [Path]
                  }

drvHash :: Derivation -> Hash
outHash :: Derivation -> Hash
instantiate :: Derivation -> IO Path
build :: Path -> IO ()
```

Let's organise this

Let's organise this

```
{ gcc = derivation { ... };

  coreutils = derivation { ... };

  python = derivation { ... };

  ...
}
```

This is a repository.

We can also store metainformation

```
{ gcc = {
    d = derivation { ... };

    description = "GNU Compiler Collection";
    version = "10.3";
    license = "GPL";
    homepage = "https://gcc.gnu.org";
  };
  ...
}
```

This is a package.

We can also store metainformation

```
{ gcc = {
    d = derivation { ... };

    description = "GNU Compiler Collection";
    version = "10.3";
    license = "GPL";
    homepage = "https://gcc.gnu.org";
  };
  ...
}
```

This is a package.

We can also store build scripts

```
{ gcc = myDerivation {
    url = "https://ftp.gnu.org/gnu/...";
    sha256 = "0i6378ig6h397zkhd7...";

    buildScript = ''
./configure --out-path=$out
make
make install
    '';
  };
  ...
}
```

We can also store build scripts

```
{ gcc = myDerivation {
    url = "https://ftp.gnu.org/gnu/...";
    sha256 = "0i6378ig6h397zkhd7...";

    buildScript = ''
./configure --out-path=$out
make
make install
make test
    '';
  };
  ...
}
```

Result: nixpkgs

Nixpkgs is a huge repository of packages.

You're not supposed to use the `derivation` function - `stdenv.mkDerivation` builds on top of it.

How do we use a repository?

How do we use a repository?

```
repo: [ repo.gcc repo.firefox ]
```

This starts to look like **configuration management**.

Configuration management

```
repo: {
  packages = [ repo.gcc repo.firefox ]

  resolvconf = "nameserver 8.8.8.8";

}
```

A derivation can only modify Nix store.

Configuration management

```
repo: {
  packages = [ repo.gcc repo.firefox ]

  resolvconf = "nameserver 8.8.8.8";

}
```

A derivation can only modify Nix store.
But it can create a script
`echo "nameserver 8.8.8.8" > /etc/resolv.conf`
and return a path to this script.

NixOS

Activation script

For declarative configuration: copy files, centralize everything.

```
build_script ''echo "$(resolv_conf)" > /etc/resolv.conf''
```

Activation script

For stateful configuration: convert the desired state to instructions.

```
build_script (if sshd_enabled
              then "systemctl start sshd"
              else "systemctl stop sshd")
```

Activation script

You can access passwords from a script, without putting them in the Nix store.

```
build_script "cp /root/my_password /etc/shadow"
```

Activation script

```
/nix/store/xiwgya4ck...ffajjrf9r014h9-sudo-1.9.6
```

Activation script

```
/nix/store/xiwgya4ck...ffajjrf9r014h9-sudo-1.9.6
/nix/store/yl2mzyayk...vfy248icd72p13-sudo-1.9.7
```

Activation script

Create a wrapper for the suid bit.

```
build_script ''
rm -f /run/suid_wrappers/*
cp ${sudo}/bin/sudo /run/suid_wrappers/
chmod u+s /run/suid_wrappers/sudo
''
```

Activation script

Add a bootloader entry if it does not exist.

Adding everything together, you get "git checkout" for your system.

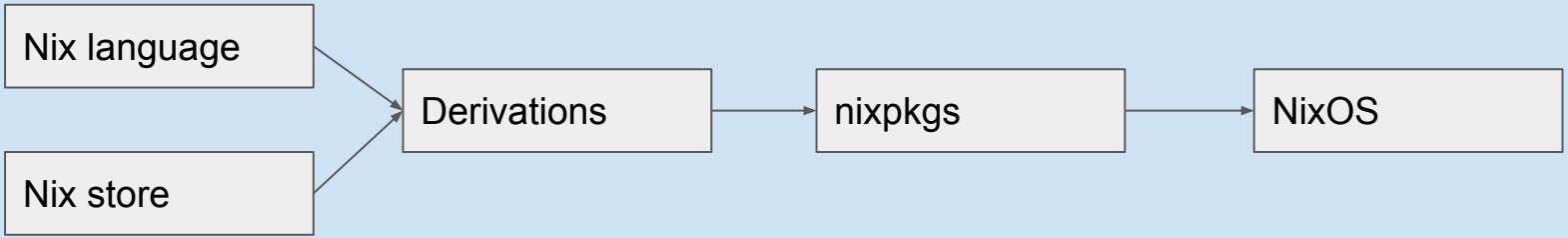
Infrastructure

Using just derivations + Nix language, we get

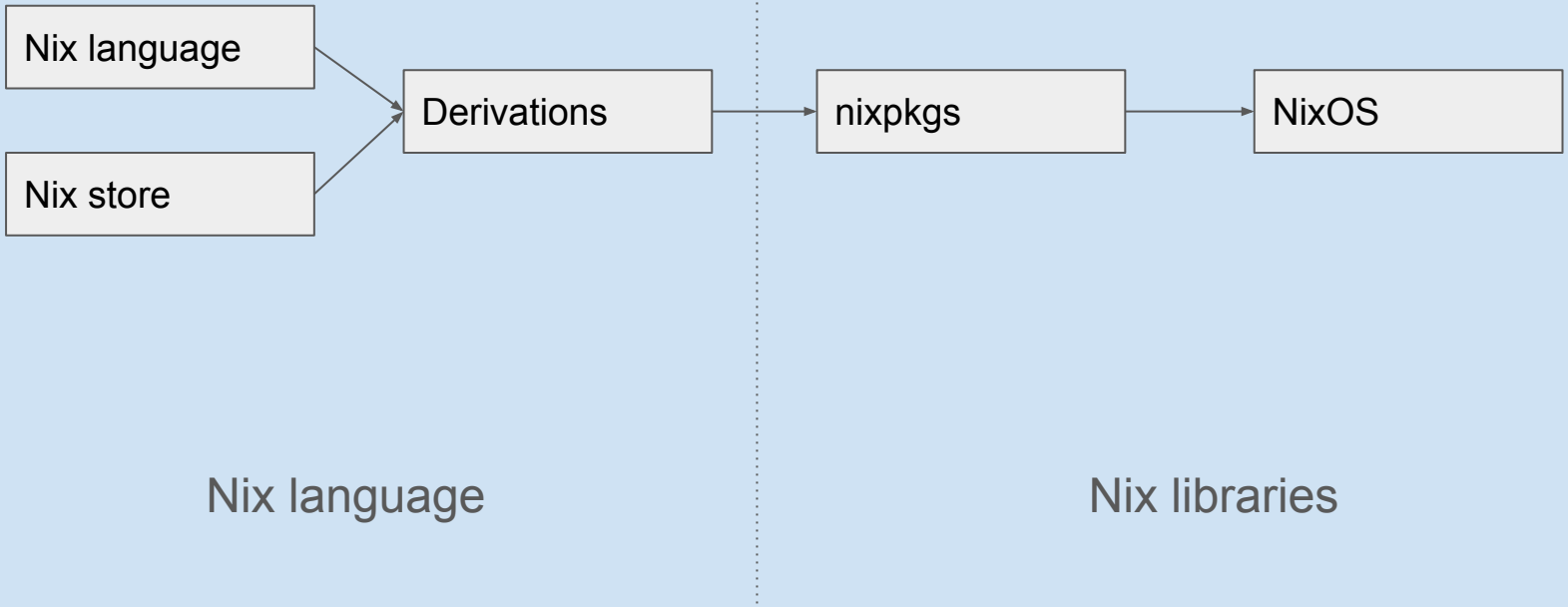
- Packages
- Repositories
- Checking for licenses
- System configuration
- Operating system

Not hardwired to Nix.
And much more: modules, types.

Layers



Layers


Nix language
Nix store
Derivations
nixpkgs
NixOS
Nix language
Nix libraries

Profiles

Every time you build a new system configuration, Nix creates a **generation**.

A profile is a collection of generations.

Generations are garbage collection roots, and the current generation is in the PATH.

Profiles can be used without NixOS.

Channels

A channel is like a git remote.

You can pull from a channel with `nix-channel --update`.

Updating NixOS:
1) Updates the channels
2) Updates nix
3) Builds a new derivation describing the
4) Runs the activation script
5) Adds a new generation

Import from derivation

```
nix-repl> derivation {
  builder = "/bin/sh";
  args = ["-c" "echo 2+2 > $out"];

  name = "write-hello";
  system = builtins.currentSystem;
}
```

Import from derivation

```
nix-repl> import (derivation {
  builder = "/bin/sh";
  args = ["-c" "echo 2+2 > $out"];

  name = "write-hello";
  system = builtins.currentSystem;
})
4
```

Import from derivation

```
nix-repl> import (derivation {
  builder = "/bin/sh";
  args = ["-c" "echo 2+2 > $out"];

  name = "write-hello";
  system = builtins.currentSystem;
})
4

unsafePerformIO :: IO a -> a
```

Import from derivation

```
nix-repl> import (derivation {
  builder = "/bin/sh";
  args = ["-c" "echo 2+2 > $out"];

  name = "write-hello";
  system = builtins.currentSystem;
})
4

join :: IO (IO a) -> IO a
```

References

nix-dev https://nix.dev/

Nix Shorts https://github.com/justinwoo/nix-shorts/

References

nix-dev https://nix.dev/

Nix Shorts https://github.com/justinwoo/nix-shorts/

Get your hands dirty!

- Install Nix locally.
  Try `nix-shell -p "haskell.packages.ghc8104.ghcWithPackages (p: [ p.aeson p.pandoc ])"`
- Build ghc with https://github.com/alpmestan/ghc.nix
- Go through Nix Pills https://nixos.org/guides/nix-pills/index.html

- Try out NixOS in a VM
- Use a USB stick and boot from it
- If you like it, install NixOS