Nix
Nix for haskellers

who heard of nix before? who is even familiar with it?

the problem with global state package managers

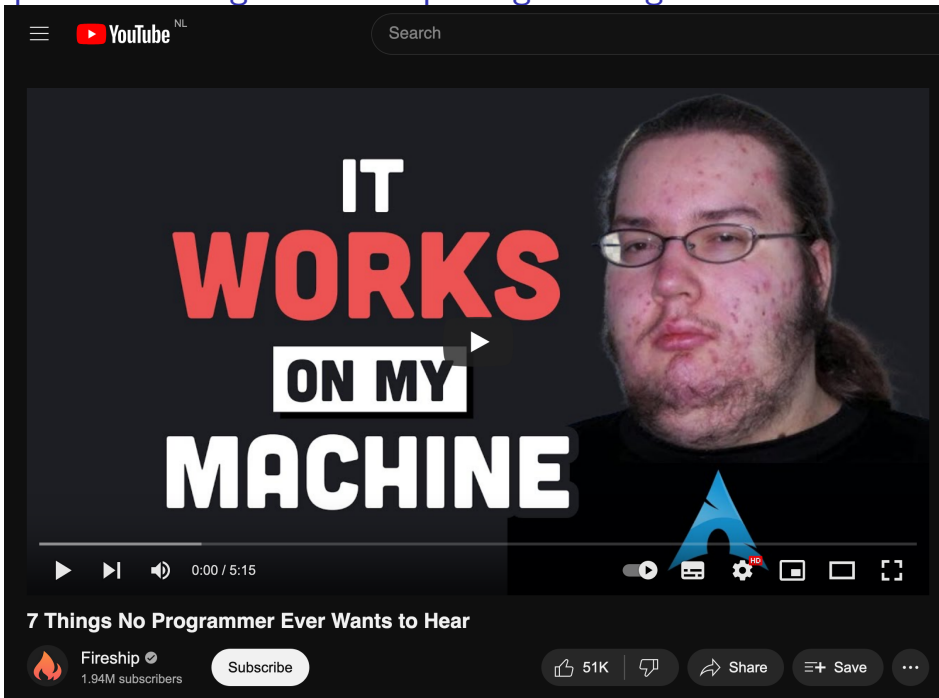
1. we want to create a haskell program.
2. we install ghc, cabal and `zlib` to our computer.
3. we add a friend to the repository.
4. their build fails.

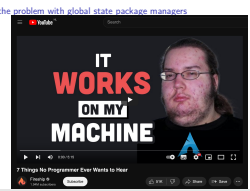
the problem with global state package managers

1. we want to create a haskell program.
2. we install ghc, cabal and `zlib` to our computer.
3. we add a friend to the repository.
4. their build fails.

so there is a global state of the installed programs and libraries. that is why i call them “global state package managers”.

the problem with global state package managers





- in other words, we cannot really help our friend.
- also, two weeks later, we need a different ghc version for a different project. but we already have ghc installed. so we have to uninstall it first.
- also, two months later, we need to build the first project again but, of course, we cannot build it anymore now either. and even worse, we do not even remember which ghc version it successfully built with before.

the problem with global state package managers

- not deterministic/reproducible
- dependency collisions

the problem with global state package managers

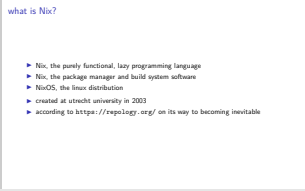
the problem with global state package managers

- not deterministic/reproducible
- dependency collisions

- ghcup solves second problem but not first.
- it does not solve the first for `zlib` either.
- crazy to have a tool that, for these specific dependencies, solves this problem that exists for any dependency, like a different compiler for example
- and that is stack's problem too, which would otherwise solve both problems.

what is Nix?

- Nix, the purely functional, lazy programming language
- Nix, the package manager and build system software
- NixOS, the linux distribution
- created at utrecht university in 2003
- according to `https://repology.org/` on its way to becoming inevitable



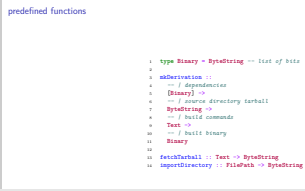
- the fact that global state causes a problem might have already tipped you off how functional programming could help
- you will see a lot of code today but you will not see any actual nix because i replaced it with pseudocode for various reasons.
  - type signatures for api
  - haskell syntax that stresses better that this really is a pure functional programming language. functions like `mkDerivation` make it quite easy to think of it as an impure configuration language. you really have to remind yourself that `mkDerivation` could be a pure function and that its implementation uses the file system is just an optimization.
  - unknown syntax is distracting and always amounts to some mental barrier

predefined functions

```haskell
type Binary = ByteString -- list of bits

mkDerivation ::
  -- | dependencies
  [Binary] ->
  -- | source directory tarball
  ByteString ->
  -- | build commands
  Text ->
  -- | built binary
  Binary

fetchTarball :: Text -> ByteString
importDirectory :: FilePath -> ByteString
```



- i will call built artifacts like executables, libraries,... “binaries”. that is easier. and often they are indeed binaries.
- so what is the task package managers or a build systems? to provide binaries. so nix needs to predefine a function returning a `Binary`. and nix calls this `mkDerivation`.
- what is a `Binary`?
- what are the ingredients for a `Binary`?
- two helper functions i will explain when using them

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

```haskell
type Binary = ByteString -- list of bits

mkDerivation ::
  -- | dependencies
  [Binary] ->
  -- | source directory tarball
  ByteString ->
  -- | build commands
  Text ->
  -- | built binary
  Binary

fetchTarball :: Text -> ByteString
importDirectory :: FilePath -> ByteString
```

```haskell
mkDerivation
  [ghc, cabal, zlib]
  (importDirectory ".")
  "cabal build && cp $(cabal list-bin exes) $out"
:: Binary
```

2023-06-17

NixNix for haskellers

```
type Binary = ByteString -- list of bits

mkDerivation ::
  -- | dependencies
  [Binary] ->
  -- | source directory tarball
  ByteString ->
  -- | build commands
  Text ->
  -- | built binary
  Binary

fetchTarball :: Text -> ByteString
importDirectory :: FilePath -> ByteString
```

```
mkDerivation
  [ghc, cabal, zlib]
  (importDirectory "./")
  "cabal build && cp $(cabal list-bin exes) $out"
:: Binary
```

explain `importDirectory`

```haskell
let
  ghc :: Binary
  ghc =
    mkDerivation
      [perl, autoconf, automake]
      (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
      "..."
  cabal :: Binary
  cabal = mkDerivation...
  zlib :: Binary
  zlib = mkDerivation...
  perl :: Binary
  perl = mkDerivation...
  autoconf :: Binary
  autoconf = mkDerivation...
  automake :: Binary
  automake = mkDerivation...
in
  mkDerivation
    [ghc, cabal, zlib]
    (importDirectory ".")
    "cabal build && cp $(cabal list-bin exes) $out"
:: Binary
```

```haskell
type Binary = ByteString -- list of bits

mkDerivation ::
  -- | dependencies
  [Binary] ->
  -- | source directory tarball
  ByteString ->
  -- | build commands
  Text ->
  -- | built binary
  Binary

fetchTarball :: Text -> ByteString
importDirectory :: FilePath -> ByteString
```

NixNix for haskellers

```
let
  ghc :: Binary
  ghc =
    mkDerivation
      [perl, autoconf, automake]
      (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
      "./.."
  cabal :: Binary
  cabal = mkDerivation...
  zlib :: Binary
  zlib = mkDerivation...
  perl :: Binary
  perl = mkDerivation...
  autoconf :: Binary
  autoconf = mkDerivation...
  automake :: Binary
  automake = mkDerivation...
in
  mkDerivation
    [ghc, cabal, zlib]
    (importDirectory "./")
    "cabal build && cp $(cabal list-bin exes) $out"
:: Binary
```

```
type Binary = ByteString -- list of bits

mkDerivation ::
  -- | dependencies
  [Binary] ->
  -- | source directory tarball
  ByteString ->
  -- | build commands
  Text ->
  -- | built binary
  Binary

fetchTarball :: Text -> ByteString
importDirectory :: FilePath -> ByteString
```

explain `fetchTarball`

how to run the binary? - Nix command reference

> *`nix-build path` - build a Nix expression*
> *The `nix-build` command builds the derivations described by the Nix expressions in path. If the build succeeds, it places a symlink to the result in the current directory.*

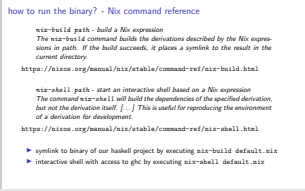
`https://nixos.org/manual/nix/stable/command-ref/nix-build.html`

> *`nix-shell path` - start an interactive shell based on a Nix expression*
> *The command `nix-shell` will build the dependencies of the specified derivation, but not the derivation itself. [...] This is useful for reproducing the environment of a derivation for development.*

`https://nixos.org/manual/nix/stable/command-ref/nix-shell.html`

- symlink to binary of our haskell project by executing `nix-build default.nix`
- interactive shell with access to ghc by executing `nix-shell default.nix`

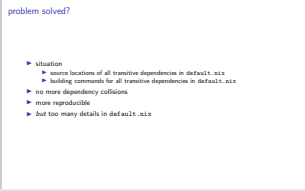
how to run the binary? - Nix command reference



1. usually, we run a binary by typing its name into our terminal, which then finds a file on the file system with that name. but all we have so far is an expression that evaluates to a `Binary`.
2. there is a "`path`" argument. so first, we need to save our nix expression to a file.
3. - `nix-build` is very nice didactically because we can very easily imagine that it just evaluates the expression in `default.nix` and saves the resulting binary to the file system.
   - `nix-shell` is a bit weird because it somehow does not evaluate the expression in `default.nix` but the first argument, of the function application that is our expression. so maybe i simplified too much when saying `mkDerivation` returns a single `Binary`.

problem solved?

- situation
  - source locations of all transitive dependencies in `default.nix`
  - building commands for all transitive dependencies in `default.nix`
- no more dependency collisions
- more reproducible
- *but* too many details in `default.nix`



we specify in detail how to build which source version of all transitive dependencies.

```
let
  ghc :: Binary
  ghc =
    mkDerivation
      [perl, autoconf, automake]
      (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
      "..."
  cabal :: Binary
  cabal = mkDerivation...
  zlib :: Binary
  zlib = mkDerivation...
  perl :: Binary
  perl = mkDerivation...
  autoconf :: Binary
  autoconf = mkDerivation...
  automake :: Binary
  automake = mkDerivation...
in
  mkDerivation
    [ghc, cabal, zlib]
    (importDirectory ".")
    "cabal build && cp $(cabal list-bin exes) $out"
:: Binary
```

```
type Binary = ByteString -- list of bits

mkDerivation ::
  -- | dependencies
  [Binary] ->
  -- | source directory tarball
  ByteString ->
  -- | build commands
  Text ->
  -- | built binary
  Binary

fetchTarball :: Text -> ByteString
importDirectory :: FilePath -> ByteString
```

```
let
  pkgs :: Map Text Binary
  pkgs = M.fromList
    [
      (
        "ghc",
        mkDerivation
          [pkgs ! "perl", pkgs ! "autoconf", pkgs ! "automake"]
          (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
          "..."
      ),
      ("cabal", mkDerivation...),
      ("zlib", mkDerivation...),
      ("perl", mkDerivation...),
      ("autoconf", mkDerivation...),
      ("automake", mkDerivation...)
    ]
in
  mkDerivation
    [pkgs ! "ghc", pkgs ! "cabal", pkgs ! "zlib"]
    (importDirectory ".")
    "cabal build && cp $(cabal list-bin exes) $out"
:: Binary
```

NixNix for haskellers

```
let
  pkgs :: Map Text Binary
  pkgs = M.fromList
    [
      (
        "ghc",
        mkDerivation
          [pkgs ! "perl", pkgs ! "autoconf", pkgs ! "automake"]
          (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
          "..."
      ),
      ("cabal", mkDerivation...),
      ("zlib", mkDerivation...),
      ("perl", mkDerivation...),
      ("autoconf", mkDerivation...),
      ("automake", mkDerivation...)
    ]
in
  mkDerivation
    [pkgs ! "ghc", pkgs ! "cabal", pkgs ! "zlib"]
    (importDirectory "./")
    "cabal build && cp $(cabal list-bin exes) $out"
  :: Binary
```

intermediate step

default.nix

```
let
  pkgs :: Map Text Binary
  pkgs =
    interpret
      (fetchTarball
        "https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz"
      )
in
  mkDerivation
    [pkgs ! "ghc", pkgs ! "cabal", pkgs ! "zlib"]
    (importDirectory ".")
    "cabal build && cp $(cabal list-bin exes) $out"
:: Binary
```

default.nix

```
let
  pkgs :: Map Text Binary
  pkgs =
    interpret
      (fetchTarball
        "https://github.com/..."
      )
in
  mkDerivation
    [pkgs ! "ghc", pkgs ! ...]
    (importDirectory ".")
    "cabal build && cp $(cab..."
:: Binary
```

https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz

```
let
  pkgs :: Map Text Binary
  pkgs = M.fromList
    [
      (
        "ghc",
        mkDerivation
          [pkgs ! "perl", pkgs ! "autoconf", pk
          (fetchTarball "https://downloads.hask
          "..."
      ),
      ("cabal", mkDerivation...),
      ("zlib", mkDerivation...),
      ("perl", mkDerivation...),
      ("autoconf", mkDerivation...),
      ("automake", mkDerivation...)
    ]
in pkgs
:: Map Text Binary
```

- why does this make sense? because these definitions are useful for anyone with a haskell project. even more, we can put more definitions in there, thousands more, every binary we know how to build. and these definitions will then be useful for anyone with any project.
- you might ask, does that not mean we `fetch` thousands of binaries? `fetchTarball` downloads the expression as a `Text`, not a `Map` of `Binary`s.

https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz

```
"let\
\  pkgs :: Map Text Binary\
\  pkgs = M.fromList\
\    [\
\      (\
\        \"ghc\",\
\        mkDerivation\
\          [pkgs ! \"perl\", pkgs ! \"autoconf\", pkgs ! \
\          (fetchTarball \"https://downloads.haskell.org/g
\          \"\"\
\      ),\
\      (\"cabal\", mkDerivation [] \"\" \"\"),\
\      (\"zlib\", mkDerivation [] \"\" \"\"),\
\      (\"perl\", mkDerivation [] \"\" \"\"),\
\      (\"autoconf\", mkDerivation [] \"\" \"\"),\
\      (\"automake\", mkDerivation [] \"\" \"\")\
\    ]\
\in pkgs\
\:: Map Text Binary\
\"
:: Text
```

default.nix

```
let
  pkgs :: Map Text Binary
  pkgs =
    interpret
      (fetchTarball
        "https://github.com/..."
      )
in
  mkDerivation
    [pkgs ! "ghc", pkgs ! ...]
    (importDirectory ".")
    "cabal build && cp $(cab..."
:: Binary
```

https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz

```
let
  pkgs :: Map Text Binary
  pkgs = M.fromList
    [
      (
        "ghc",
        mkDerivation
          [pkgs ! "perl", pkgs ! "autoconf", pk
          (fetchTarball "https://downloads.hask
          "..."
      ),
      ("cabal", mkDerivation...),
      ("zlib", mkDerivation...),
      ("perl", mkDerivation...),
      ("autoconf", mkDerivation...),
      ("automake", mkDerivation...)
    ]
in pkgs
:: Map Text Binary
```

1. – that is while i need `interpret` here.
   – so next, you might ask, ok, we do not download thousands of binaries but does `interpret` not *build* thousands of binaries? no because lazy.
2. – speaking of the `fetchTarball` function, we saw it before to fetch source directories, is this a pure function or does it violate referential transparency?
   – why does it matter by the way? a build is reproducible if and only if its expression evaluates to the same value every time. so reproducibility of the package manager corresponds to referential transparency of the language.
   – but `fetchTarball` unfortunately does violate referential transparency

https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz

```
let
  pkgs :: Map Text Binary
  pkgs = M.fromList
    [
      (
        "ghc",
        mkDerivation
          [pkgs ! "perl", pkgs ! "autoconf", pkgs ! "automake"]
          (
            fetchTarball
              "https://downloads.haskell.org/ghc-9.4.3-src.tar"
              "eaf63949536ede50ee39179f2299d5094eb9152d87cc6fb2175006bc98e8905a"
          )
          "..."
      ),
      ("cabal", mkDerivation...),
      ("zlib", mkDerivation...),
      ("perl", mkDerivation...)
    ]
in pkgs
:: Map Text Binary
```

https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz

```
let
  pkgs :: Map Text Binary
  pkgs = M.fromList
    [ (
      "ghc",
      mkDerivation
        [pkgs ! "perl", pkgs ! "autoconf", pkgs ! "automake"]
        (
          fetchTarball
            "https://downloads.haskell.org/ghc-9.4.3-src.tar"
            "sha256:4953536ede50ae39179f2299d5094eb9152d87ccfdb2175006bc98e8905a"
        )
        "..."
    ),
    ("cabal", mkDerivation...),
    ("zlib", mkDerivation...),
    ("perl", mkDerivation...)
    ]
in pkgs
:: Map Text Binary
```

so `fetchTarball` actually requires a second argument, which is a hash of the tarball. so `fetchTarball` can only successfully evaluate to one value. otherwise, nix will detect the referential transparency violation by comparing hashes and terminate immediately. so you could argue that impurity cannot be observed from within the language. also, when our friend tries to build our haskell project and the online source of one of the dependencies has changed, nix can report which one it is, my friend can tell me, and we can investigate.

default.nix

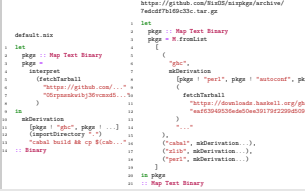
```
let
  pkgs :: Map Text Binary
  pkgs =
    interpret
      (fetchTarball
        "https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz"
        "05rpnsnkwibj36vcmxd55ms2brl3clbi5gh5cnks6qaw2x6mdsag"
      )
in
  mkDerivation
    [pkgs ! "ghc", pkgs ! "cabal", pkgs ! "zlib"]
    (importDirectory ".")
    "cabal build && cp $(cabal list-bin exes) $out"
:: Binary
```

default.nix

```
let
  pkgs :: Map Text Binary
  pkgs =
    interpret
      (fetchTarball
        "https://github.com/..."
        "05rpnsnkwibj36vcmxd5..."
      )
in
  mkDerivation
    [pkgs ! "ghc", pkgs ! ...]
    (importDirectory ".")
    "cabal build && cp $(cab..."
:: Binary
```

https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz

```
let
  pkgs :: Map Text Binary
  pkgs = M.fromList
    [
      (
        "ghc",
        mkDerivation
          [pkgs ! "perl", pkgs ! "autoconf", pk
          (
            fetchTarball
              "https://downloads.haskell.org/gh
              "eaf63949536ede50ee39179f2299d509
          )
          "..."
      ),
      ("cabal", mkDerivation...),
      ("zlib", mkDerivation...),
      ("perl", mkDerivation...)
    ]
in pkgs
:: Map Text Binary
```



- notice how this left hash actually depends transitively on the content of the online source code. this left hash depends on this right file and therefore this right hash, and this right hash depends on the content of the online source.
- so far we seem to need to rebuild ghc and all transitive dependencies every time we restart our computer.
- we need some kind of sharing that outlives restarts. caching.
- notice that we can compute a hash for a binary before building it just from `mkDerivation`'s arguments... different binaries, have different nix expressions, which have different hashes.

caching/sharing

1. Nix can compute a binaries hash before building it.
2. lookup by that hash in a local cache for built binaries
3. lookup by that hash in an online cache for built binaries
4. otherwise, build and cache by that hash.

- no indeterminism via false cache hits thanks to cryptographic SHA 256 hashing
- no dependency collision thanks to cryptographic SHA 256 hashing
- secure sharing of local cache between different users thanks to cryptographic SHA 256 hashing

caching/sharing

1. Nix can compute a binaries hash before building it.
2. lookup by that hash in a local cache for built binaries
3. lookup by that hash in an online cache for built binaries
4. otherwise, build and cache by that hash.

- no indeterminism via false cache hits thanks to cryptographic SHA 256 hashing
- no dependency collision thanks to cryptographic SHA 256 hashing
- secure sharing of local cache between different users thanks to cryptographic SHA 256 hashing

did we just reintroduce global state?

conclusion

- a functional programming language can be a package manager and build system
- reproducibility of the package manager corresponds to referential transparency in the language
- caching of the package manager corresponds to sharing in the language
- i do not know what avoiding dependency collision corresponds to