



2023-06-17

NixNix 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

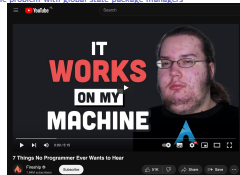
1. we want to create a haskell program.
2. we install ghc, cabal and x11h to our computer.
3. we add a friend to the repository.
4. their build fails.

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

The image shows a YouTube video player interface. At the top left, there is a menu icon and the YouTube logo with 'NL' next to it. A search bar is located at the top center. The video content area features a dark background with a person's face on the right side. Overlaid on the left is the text 'IT WORKS ON MY MACHINE' in large, bold, white and red letters. Below the video area is a playback progress bar showing '0:00 / 5:15' and various control icons like play, volume, and settings. At the bottom, the video title '7 Things No Programmer Ever Wants to Hear' is displayed, along with the channel name 'Fireship' and '1.94M subscribers'. There are also buttons for 'Subscribe', 'Like' (51K), 'Share', and 'Save'. The bottom right corner shows navigation icons and the page number '3 / 21'.

└ 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

- ▶ 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

└ 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://reprology.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

```
1  type Binary = ByteString -- list of bits
2
3  mkDerivation ::
4    -- | dependencies
5    [Binary] ->
6    -- | source directory tarball
7    ByteString ->
8    -- | build commands
9    Text ->
10   -- | built binary
11   Binary
12
13  fetchTarball :: Text -> ByteString
14  importDirectory :: FilePath -> ByteString
```

└ predefined functions

```
1 type Binary = ByteString -- list of bits
2
3 mkDerivation ::
4   -- / dependencies
5   [Binary] ->
6   -- / source directory tarball
7   ByteString ->
8   -- / build commands
9   Text ->
10  -- / built binary
11  Binary
12
13 fetchTarball :: Text -> ByteString
14 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

```
1  type Binary = ByteString -- list of bits
2
3  mkDerivation ::
4      -- | dependencies
5      [Binary] ->
6      -- | source directory tarball
7      ByteString ->
8      -- | build commands
9      Text ->
10     -- | built binary
11     Binary
12
13  fetchTarball :: Text -> ByteString
14  importDirectory :: FilePath -> ByteString
```

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

```
1 type Binary = ByteString -- list of bits
2
3 mkDerivation ::
4   -- / dependencies
5   [Binary] ->
6   -- / source directory tarball
7   ByteString ->
8   -- / build commands
9   Text ->
10  -- / build binary
11  Binary
12
13 mkDerivation
14 [shc, cabal, alib]
15 (importDirectory ".")
16 "cabal build && cp $(cabal list-bin exe) $out"
17 :: Binary
```

explain `importDirectory`

```

1  let
2    ghc :: Binary
3    ghc =
4      mkDerivation
5        [perl, autoconf, automake]
6        (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
7        "...")
8    cabal :: Binary
9    cabal = mkDerivation...
10   zlib :: Binary
11   zlib = mkDerivation...
12   perl :: Binary
13   perl = mkDerivation...
14   autoconf :: Binary
15   autoconf = mkDerivation...
16   automake :: Binary
17   automake = mkDerivation...
18  in
19    mkDerivation
20      [ghc, cabal, zlib]
21      (importDirectory ".")
22      "cabal build && cp $(cabal list-bin exes) $out"
23  :: Binary

```

```

1  type Binary = ByteString -- list of bits
2
3  mkDerivation ::
4    -- / dependencies
5    [Binary] ->
6    -- / source directory tarball
7    ByteString ->
8    -- / build commands
9    Text ->
10   -- / built binary
11   Binary
12
13  fetchTarball :: Text -> ByteString
14  importDirectory :: FilePath -> ByteString

```

```

1 let
2   ghc :: Binary
3   ghc =
4     mkDerivation
5     [perl, autoconf, automake]
6     (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
7     ""
8   cabal :: Binary
9   cabal = mkDerivation...
10  zlib :: Binary
11  zlib = mkDerivation...
12  perl :: Binary
13  perl = mkDerivation...
14  autoconf :: Binary
15  autoconf = mkDerivation...
16  automake :: Binary
17  automake = mkDerivation...
18  in
19  mkDerivation
20  [ghc, cabal, zlib]
21  (importDirectory ".")
22  "cabal build && cp $(cabal list-bin exe) $out"
23 :: Binary

```

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

```

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

```

- 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**.
- there is a “path” argument. so first, we need to save our nix expression to a file.
- `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`

└ problem solved?

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.

```

1  let
2      ghc :: Binary
3      ghc =
4          mkDerivation
5              [perl, autoconf, automake]
6              (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
7              "...")
8      cabal :: Binary
9      cabal = mkDerivation...
10     zlib :: Binary
11     zlib = mkDerivation...
12     perl :: Binary
13     perl = mkDerivation...
14     autoconf :: Binary
15     autoconf = mkDerivation...
16     automake :: Binary
17     automake = mkDerivation...
18  in
19     mkDerivation
20         [ghc, cabal, zlib]
21         (importDirectory ".")
22         "cabal build && cp $(cabal list-bin exes) $out"
23     :: Binary

```

```

1  type Binary = ByteString -- list of bits
2
3  mkDerivation ::
4      -- / dependencies
5      [Binary] ->
6      -- / source directory tarball
7      ByteString ->
8      -- / build commands
9      Text ->
10     -- / built binary
11     Binary
12
13     fetchTarball :: Text -> ByteString
14     importDirectory :: FilePath -> ByteString

```

```
1  let
2    pkgs :: Map Text Binary
3    pkgs = M.fromList
4      [
5        (
6          "ghc",
7          mkDerivation
8            [pkgs ! "perl", pkgs ! "autoconf", pkgs ! "automake"]
9            (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
10           "...")
11        ),
12        ("cabal", mkDerivation...),
13        ("zlib", mkDerivation...),
14        ("perl", mkDerivation...),
15        ("autoconf", mkDerivation...),
16        ("automake", mkDerivation...)
17      ]
18  in
19    mkDerivation
20      [pkgs ! "ghc", pkgs ! "cabal", pkgs ! "zlib"]
21      (importDirectory ".")
22      "cabal build && cp $(cabal list-bin exes) $out"
23  :: Binary
```

```
1 let
2   pkgs = Map Text Binary
3   pkgs = H.fromList
4   [
5     {
6       "ghc",
7       sdDerivation
8       [pkgs ! "perl", pkgs ! "autoconf", pkgs ! "automake"]
9       (fetchTarball "https://downloads.haskell.org/ghc-9.4.3-src.tar")
10      ...
11    },
12    ("cabal", sdDerivation...),
13    ("zlib", sdDerivation...),
14    ("perl", sdDerivation...),
15    ("autoconf", sdDerivation...),
16    ("automake", sdDerivation...)
17  ]
18 in
19 sdDerivation
20 [pkgs ! "ghc", pkgs ! "cabal", pkgs ! "zlib"]
21 (importDirectory ".")
22 "cabal build --up $(cabal list-bin exec) $out"
23 :: Binary
```

intermediate step

default.nix

```
1  let
2    pkgs :: Map Text Binary
3    pkgs =
4      interpret
5        (fetchTarball
6          "https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz"
7        )
8  in
9    mkDerivation
10     [pkgs ! "ghc", pkgs ! "cabal", pkgs ! "zlib"]
11     (importDirectory ".")
12     "cabal build && cp $(cabal list-bin exes) $out"
13  :: Binary
```


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

default.nix

```
1 let
2   pkgs :: Map Text Binary
3   pkgs =
4     interpret
5       (fetchTarball
6         "https://github.com/..."
7       )
8 in
9   mkDerivation
10    [pkgs ! "ghc", pkgs ! ...]
11    (importDirectory ".")
12    "cabal build && cp $(cab..."
13  :: Binary
```

```
1 let
2   pkgs :: Map Text Binary
3   pkgs = M.fromList
4     [
5       (
6         "ghc",
7         mkDerivation
8           [pkgs ! "perl", pkgs ! "autoconf", pk
9           (fetchTarball "https://downloads.hask
10            "...")
11        ),
12        ("cabal", mkDerivation...),
13        ("zlib", mkDerivation...),
14        ("perl", mkDerivation...),
15        ("autoconf", mkDerivation...),
16        ("automake", mkDerivation...)
17      ]
18 in pkgs
19  :: Map Text Binary
```

```

https://github.com/NixOS/nixpkgs/archive/
7edcd7b169c33c.tar.gz
default.nix
let
  page :: Map Text Binary
in
  page = M.fromList
    [
      ("ghc",
        mkDerivation
          [page ! "perl", page ! "autoconf", pk
            fetchTarball "https://www.gnu.org/software/autoconf/..."]
      )
    ]
in
  mkDerivation
    [page ! "ghc", page ! ...]
    ("lib", mkDerivation...)
    ("importDirectory", ...)
    ("perl", mkDerivation...)
    ("autoconf", mkDerivation...)
  :: Binary
in page
:: 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.

```
1  "let\  
2  \  pkgs :: Map Text Binary\  
3  \  pkgs = M.fromList\  
4  \    [\  
5  \      (\  
6  \        \"ghc\",\  
7  \        mkDerivation\  
8  \          [pkgs ! \"perl\", pkgs ! \"autoconf\", pkgs ! \  
9  \            (fetchTarball \"https://downloads.haskell.org/g  
10 \            \"\"\  
11 \          ),\  
12 \          (\"cabal\", mkDerivation [] \"\" \"\"),\  
13 \          (\"zlib\", mkDerivation [] \"\" \"\"),\  
14 \          (\"perl\", mkDerivation [] \"\" \"\"),\  
15 \          (\"autoconf\", mkDerivation [] \"\" \"\"),\  
16 \          (\"automake\", mkDerivation [] \"\" \"\"))\  
17 \    ]\  
18 \in pkgs\  
19 \:: Map Text Binary\  
20 \  
21 :: Text
```

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

default.nix

```
1 let
2   pkgs :: Map Text Binary
3   pkgs =
4     interpret
5       (fetchTarball
6         "https://github.com/..."
7       )
8 in
9   mkDerivation
10    [pkgs ! "ghc", pkgs ! ...]
11    (importDirectory ".")
12    "cabal build && cp $(cab..."
13  :: Binary
```

```
1 let
2   pkgs :: Map Text Binary
3   pkgs = M.fromList
4     [
5       (
6         "ghc",
7         mkDerivation
8           [pkgs ! "perl", pkgs ! "autoconf", pk
9           (fetchTarball "https://downloads.hask
10            "...")
11        ),
12        ("cabal", mkDerivation...),
13        ("zlib", mkDerivation...),
14        ("perl", mkDerivation...),
15        ("autoconf", mkDerivation...),
16        ("automake", mkDerivation...)
17      ]
18 in pkgs
19  :: Map Text Binary
```

```

https://github.com/NixOS/nixpkgs/archive/
7edcd7b169c33c.tar.gz
default.nix
let
  pkg = M.fromList
  pkg :: Map Text Binary
in
  pkg =
    {
      "ghc",
      mkDerivation
        [page ! "perl", page ! "autoconf", pk
          (fetchTarball "https://www.gnu.org/...")
          (fetchTarball "https://www.gnu.org/...")
        ]
    }
  mkDerivation
    [page ! "ghc", page ! ...]
    (importDirectory ".")
    "cabal build --cp $(cabal...)"
    ("autoconf", mkDerivation...)
  in
    [
      in page
    ]
  :: Map Text Binary

```

- 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.
- 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

```
1  let
2    pkgs :: Map Text Binary
3    pkgs = M.fromList
4      [
5        (
6          "ghc",
7          mkDerivation
8            [pkgs ! "perl", pkgs ! "autoconf", pkgs ! "automake"]
9            (
10             fetchTarball
11               "https://downloads.haskell.org/ghc-9.4.3-src.tar"
12               "eaf63949536ede50ee39179f2299d5094eb9152d87cc6fb2175006bc98e8905a"
13             )
14             "...")
15        ),
16        ("cabal", mkDerivation...),
17        ("zlib", mkDerivation...),
18        ("perl", mkDerivation...)
19      ]
20  in pkgs
21  :: Map Text Binary
```

```
https://github.com/NixOS/nixpkgs/archive/7edcdd7b169c33c.tar.gz
1 let
2   pkgs = import pkgs {
3     pkgs = N.Frodo;
4   };
5   {
6     "ghc",
7     mkDerivation
8     {
9       [pkgs ! "perl", pkgs ! "autoconf", pkgs ! "automake"]
10      fetchTarball
11      "https://download.haskell.org/ghc-9.4.3-src.tar"
12      "aefc3949c36e4e0aa35179f22949504e4b01c087c0f52175006bc9e4905a"
13    }
14    "...",
15  },
16  ("cabal", mkDerivation...),
17  ("nix", mkDerivation...),
18  ("perl", mkDerivation...)
19 }
20 in pkgs
21 -- Hap 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

```
1  let
2    pkgs :: Map Text Binary
3    pkgs =
4      interpret
5        (fetchTarball
6          "https://github.com/NixOS/nixpkgs/archive/7edcdf7b169c33c.tar.gz"
7          "05rpnsnkwiwj36vcmx55ms2brl3clbi5gh5cnks6qaw2x6mdsag"
8        )
9  in
10   mkDerivation
11     [pkgs ! "ghc", pkgs ! "cabal", pkgs ! "zlib"]
12     (importDirectory ".")
13     "cabal build && cp $(cabal list-bin exes) $out"
14  :: Binary
```


default.nix

```
1 let
2   pkgs :: Map Text Binary
3   pkgs =
4     interpret
5       (fetchTarball
6         "https://github.com/..."
7         "05rpnsnkwibj36vcmd5...")
8
9 in
10  mkDerivation
11    [pkgs ! "ghc", pkgs ! ...]
12    (importDirectory ".")
13    "cabal build && cp $(cab..."
14  :: Binary
```

```
1 let
2   pkgs :: Map Text Binary
3   pkgs = M.fromList
4     [
5       (
6         "ghc",
7         mkDerivation
8           [pkgs ! "perl", pkgs ! "autoconf", pk
9           (
10            fetchTarball
11              "https://downloads.haskell.org/gh
12              "eaf63949536ede50ee39179f2299d509
13            )
14            "...")
15       ),
16       ("cabal", mkDerivation...),
17       ("zlib", mkDerivation...),
18       ("perl", mkDerivation...)
19     ]
20 in pkgs
21 :: Map Text Binary
```

```

https://github.com/NixOS/nixpkgs/archive/
2e2cd7b1e9c3c.tar.gz
default.nix
let
  pkgs = Map Text Binary
in
let
  pkgs = M.fromList
  [
    {
      ghc =
        mkDerivation
        {
          fetchTarball =
            (
              "https://github.com/..."
            )
            "ghc"
            "https://download.haskell.org/ghc/..."
        }
    }
  ]
in
mkDerivation
{
  (pkgs ! "ghc", pkgs ! ...)
  (importDirectory "...")
  ("cabal build & cp $cab..."
  ("nix", 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