A Tale of Nix and Nickel

YOW! Lambda Jam

Yann Hamdaoui

May 5, 2021

TWEAG

Introduction

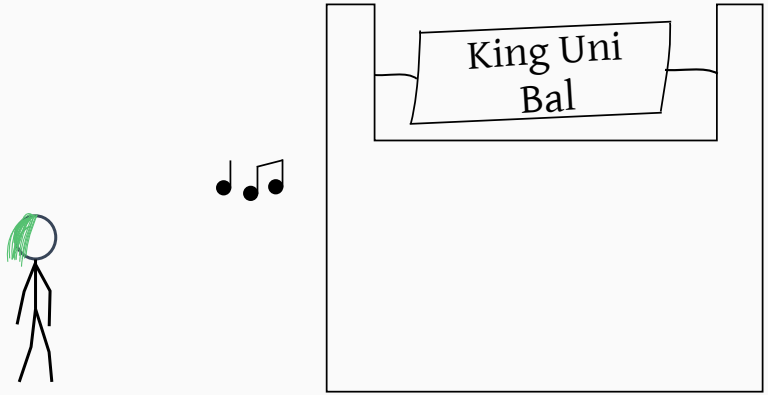
A cautionary tale

Once upon a time…

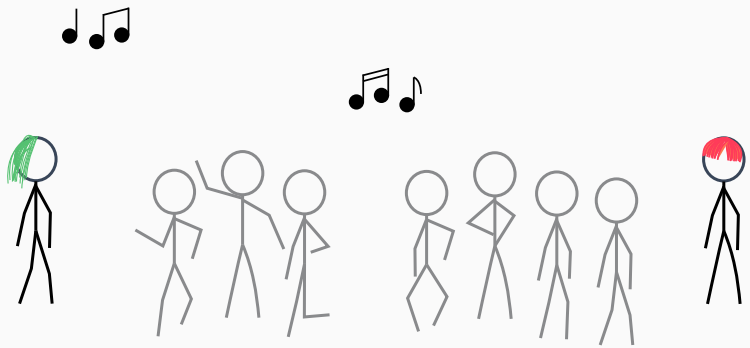
A cautionary tale

There was a brilliant princess

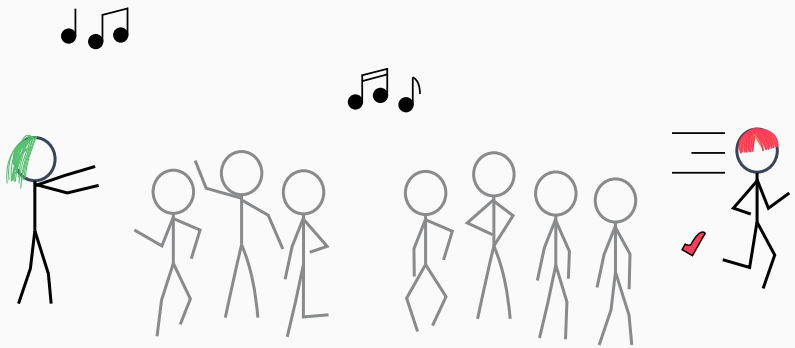
One night, she goes to the Kingdom University Bal



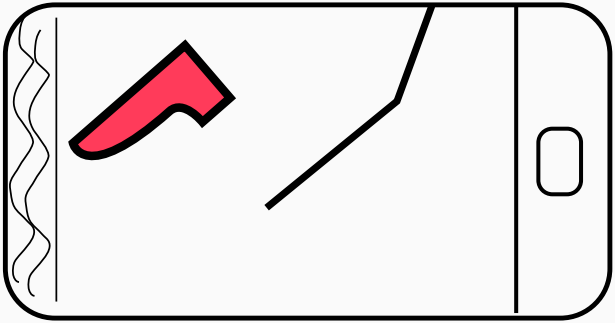
Where she falls in love with a mysterious girl



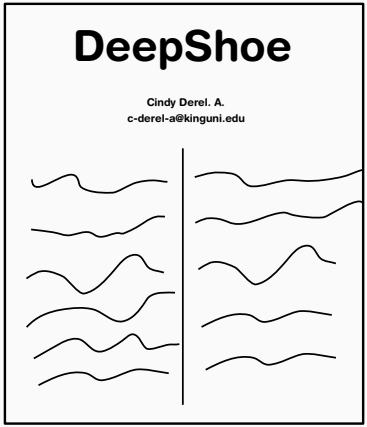
But when midnight strikes, the girl suddenly runs!

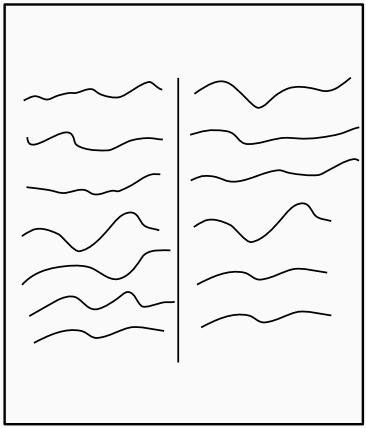


She could only get a photo of her shoe

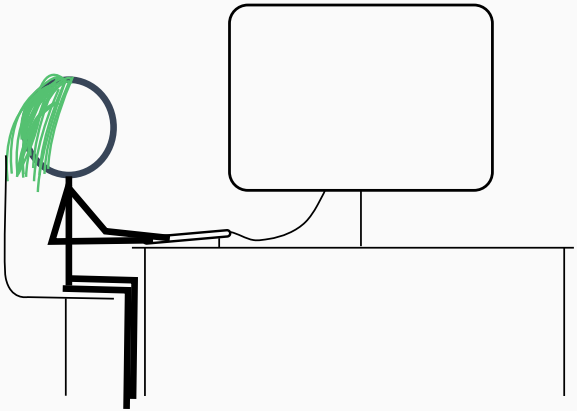


Mercifully, there is the DeepShoe paper!





She downloads the repository, which requires Python



Well, Python2, actually

$python main.py
Failed to import the shoe module...

$install python2

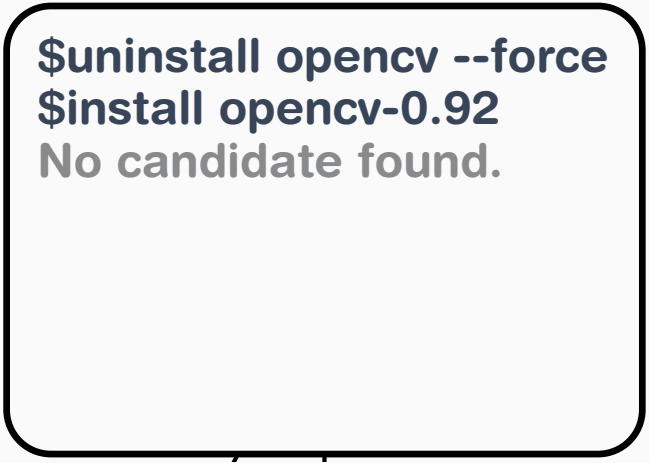
And the python bindings to opencv as well

```
$python2 main.py
Failed to import the
opencv module

$pip2 install opencv-pyth
on
```

With opencv v1.14

```
$python2 main.py
Undefined symbol:
  "_ZNreadImgRaw12"

$pack version opencv
package: OpenCV
version: 2.0
```

She moves heaven and earth



To find a solution

Without success

```
$git clone ...opencv.git .
$git checkout rel-0.92
$make install

$python2 main.py
Segmentation fault
```

A cautionary tale

The end

works on my machine ¯_(ツ)_/¯

works on my machine ¯_(ツ)_/¯

Reproducibility

works on my machine ¯_(ツ)_/¯

Reproducibility

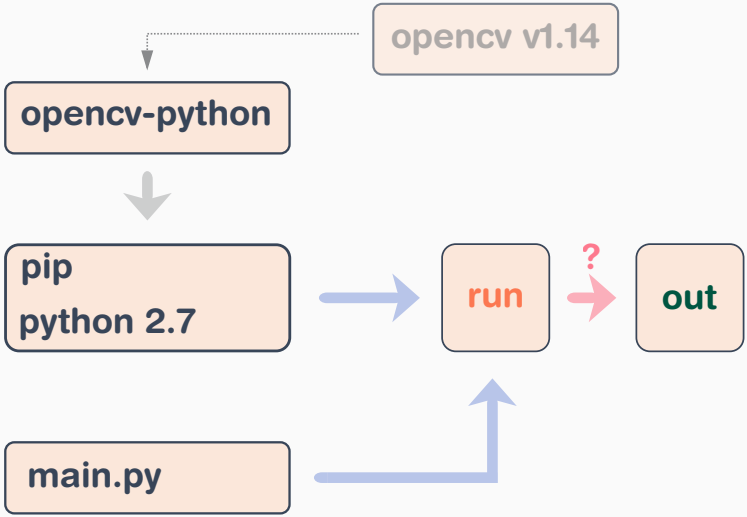
1. Concrete and widespread

works on my machine ¯_(ツ)_/¯

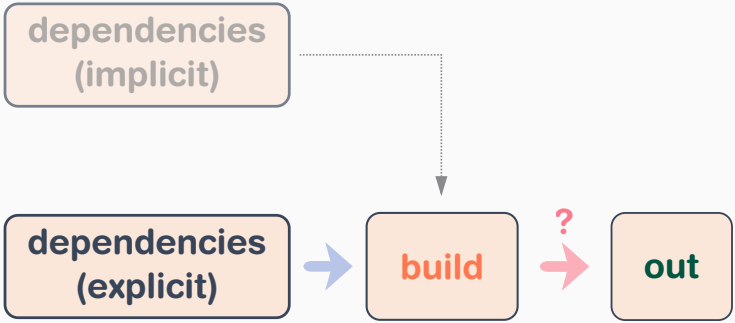
Reproducibility

1. Concrete and widespread
2. Not addressed by mainstream tools

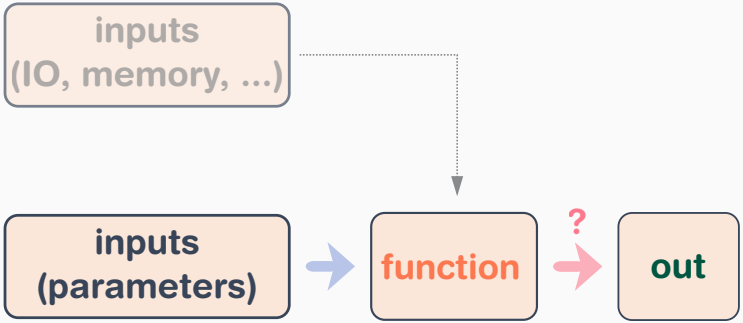
The problem



Looks familiar?



Looks familiar?



Functional approach to reproducibility

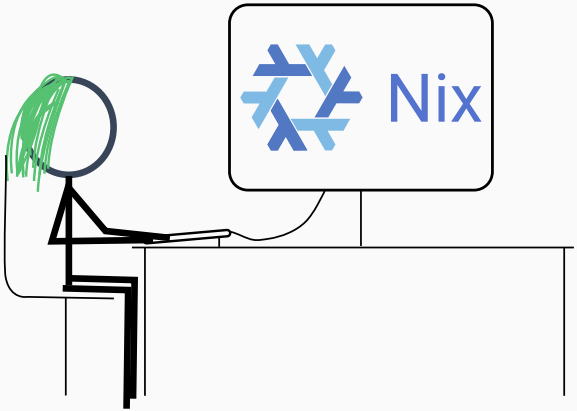
Nix: the (pure) functional package manager

works on my machine ¯_(ツ)_/¯

Nix

works ~~on my machine ¯_(ツ)_/¯~~
everywhere

What if the princess had access to a version packaged by Nix?



Steps

Steps

1. Describe a package and its dependencies in full

Describing

```
{python2WithOpenCV, opencv, stdenv}:
stdenv.mkDerivation rec {
  pname = "gh-from-shoe";
  version = "2021-04-30";

  buildInputs = [ python2WithOpenCV opencv ];

  installPhase = ''
    mkdir -p $out/bin
    cp ${./main.py} $out/bin/gh-from-shoe
  '';
};
```

gh-from-shoe/default.nix

Describing

```
Derive(
  [("out","/nix/store/qya..-gh-from-shoe","","")],
  [
    ("/nix/store/ae4..-python-2-7-10.drv",
      ["out"]),
    ("/nix/store/78f..-opencv-1-14.drv",
      ["out"]),
    ...
  ["/nix/store/9kr..-default-builder.sh"],
  "x86_64-linux",
  ...
```

gh-from-shoe-1-0.drv (generated)

Steps

1. Describe a package and its dependencies in full
2. Build it in isolation

Building

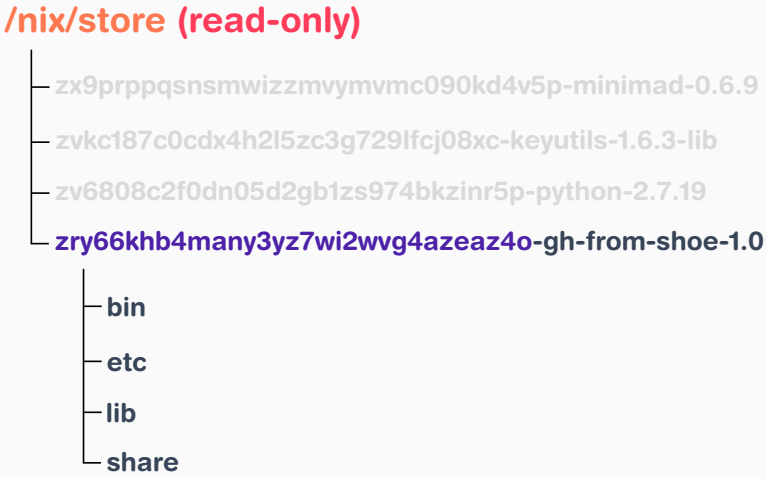
gh-from-shoe-1.0$ nix build

1. Pull and build dependencies
   (`opencv-1-14`, `python-2-7-10`, …)
2. Create an isolated environment.
3. Run the builder.

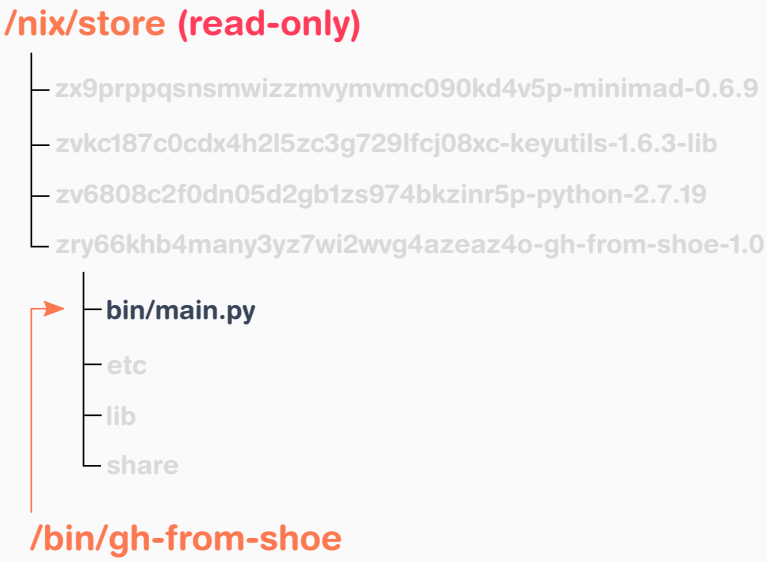
Steps

1. Describe a package and its dependencies in full
2. Build it in isolation
3. Put the result in the store

Storing


/nix/store (read-only)
zx9prppqsnsmwizzmvymvmc090kd4v5p-minimad-0.6.9
zvkc187c0cdx4h2l5zc3g729lfcj08xc-keyutils-1.6.3-lib
zv6808c2f0dn05d2gb1zs974bkzinr5p-python-2.7.19
zry66khb4many3yz7wi2wvg4azeaz4o-gh-from-shoe-1.0
bin
etc
lib
share

Storing


/nix/store (read-only)
zx9prppqsnsmwizzmvymvmc090kd4v5p-minimad-0.6.9
zvkc187c0cdx4h2l5zc3g729lfcj08xc-keyutils-1.6.3-lib
zv6808c2f0dn05d2gb1zs974bkzinr5p-python-2.7.19
zry66khb4many3yz7wi2wvg4azeaz4o-gh-from-shoe-1.0
bin/main.py
etc
lib
share
/bin/gh-from-shoe

Steps

1. Describe a package and its dependencies in full
2. Build it in isolation
3. Put the result in the store
4. Profit: find love!

Perks

- Reproducible
- Declarative
- Complete dependencies
- Fearless upgrades: atomic upgrades and rollbacks

Usage examples

- Nix: package management
- NixOS: declarative system configuration
- Nix shell: project-specific environments
- NixOps: Nix-based cloud deployment

Constraints

- Steep learning curve
- Everything has to be "Nixified"

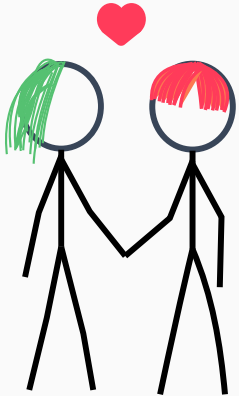
Purely functional package management

Nix	Pure functional programming
Read-only store	Immutability
Hash addressing + sharing	Hash consing
Cleaning	Garbage collection
Reproducibility	Referential transparency

Nix expressions

Epilogue

The princess found love and now wants to use Nix for her own projects



Package as a function

Building a package should be a pure function: use a functional programming language!

Nix expressions

The Nix language

JSON + λ (higher-order functions)

Nix expressions

```
{python2WithOpenCV, opencv, stdenv}:
stdenv.mkDerivation rec {
  pname = "gh-from-shoe-rust";
  version = "2021-04-30";

  buildInputs = [ python2WithOpenCV opencv ];

  installPhase = ''
    mkdir -p $out/bin
    cp ${./myscript.py} $out/bin/myscript
  '';
};
```

gh-from-shoe/default.nix

Derivation: Nix object code

```
Derive(
  [("out","/nix/store/qya..-gh-from-shoe","","")],
  [
    ("/nix/store/ae4..-python-2-7-10.drv",
      ["out"]),
    ("/nix/store/78f..-opencv-1-14.drv",
      ["out"]),
    ...
  ["/nix/store/9kr..-default-builder.sh"],
  "x86_64-linux",
  ...
```

/nix/store/27az7...gh-from-shoe-1-0.drv

State of affairs

Nix expressions outgrew their initial scope.

In the wild

- Various user-defined abstractions
- Object systems (kind of): overriding
- A module system: NixOS
- Non-trivial algorithms (e.g. topological sort)

All of this without types!

Nickel

Meet Nickel

A new take

- Gradual typing
- Run-time contracts
- Recursive records merge system
- Stand-alone language: free us from YAML templating! (Terraform, Kubernetes, build systems, etc.)

A teaser: contract

```
let Port = ...

let Service = {
  name | doc "Service name"
       | Str,

  openPorts | doc "Open ports (firewall)"
            | List #Port
            | default = [],
  ...
}
```

contracts.ncl

A teaser: configuration

```
let portToUrl : Str -> Num -> Str =
  fun host port => ...  in

{
  name = "nginx",
  openPorts = [80, 443],
  server = "localhost",
  urls = lists.map
    (portToUrl server)
    openPorts,
}
| #Service
```

nginx.ncl

A teaser: result

```
{
  "name": "nginx",
  "openPorts": [
    80,
    443
  ],
  "server": "localhost",
  "urls": [
    "http://localhost",
    "https://localhost"
  ]
}
```

nginx.json

Untyped code

By default, code is untyped:

- Terminating & fixed inputs
- Contracts for validation
- JSON interop

Example

```
services = [
  "init",
  {name = "firewall", bin = "/bin/firewall"},
  {name = "service", repo = "github.com/johndoe/dns-service"}
]
```

Heterogeneous values

Typed code

Library code is statically typed:

- Triggered by *annotations*
- Scoped
- Type-inference

Example

```
1 map : forall a b. (a -> b) -> List a -> List b
2     = fun f list =>
3   if list == [] then []
4   else
5     let head = lists.head list in
6     let tail = lists.tail list in
7     [f head] @ map f tail
```

Statically typed map

Interaction typed/untyped

Problem
Untyped code can sneak in ill-typed parameters

Example

```
1 let add : Num -> Num -> Num
2         = fun x y => x + y in
3 add "a" 0
```

```
let add : Num -> Num -> Num = fun x y => x + y
                                         ^

This expression has type Str, expected Num
```

Contracts, the invisible glue

Typed code is protected by run-time casts, or *contracts*.

```
let safeNum = fun value =>
    if builtins.isNum value then value
    else panic! in

let addSafe = fun x y =>
  let safeX = safeNum x in
  let safeY = safeNum y in
  safeNum (add safeX safeY)
```

Generated code for `add` (simplified)

Contracts, the invisible glue

```
error: Blame error: contract broken by the caller.
   ┌─ :1:1
   │
1  │ Num -> Num -> Num
   │ --- expected type of the argument provided by the caller
   │
   ┌─ repl-input-0:2:5
   │
2  │ add "a" 0
   │     --- evaluated to this expression
   │
   ┌─ <unkown> (generated by evaluation):1:1
   │
1  │ "a"
   │ --- evaluated to this value
   │
note:
   ┌─ repl-input-0:1:11
   │
1  │ let add : Num -> Num -> Num = fun x y => x + y in
   │           ^^^^^^^^^^^^^^^^^ bound here
```

First-class contracts

```
let Url = let pattern = "[-a-zA-Z0-9@:..." in
  fun label value =>
    if builtins.isStr value then
      if strings.isMatch value pattern then
        value
      else
        contracts.blame
          (contracts.tag "invalid URL" label)
    else
      contracts.blame
        (contracts.tag "not a string" label) in

let mkUrls | {url: #Url, pattern: Str} -> List #Url = ...
```

First-class contracts

```
Derivation | doc "A Nix package, in Nickel" = {
  name | Str,
  buildInputs | List #NixPackage,
},

NixPackage | doc "Interchange format" = {
  package | Str,
  input | Str
        | default = "nixpkgs",
  _type = "package",
},
```

First-class contracts

Perks

- Can check arbitrary properties
- Composable
- Allow safe typed/untyped interactions
- Built-in error reporting

Limits

- Run-time cost
- Untriggered code paths

Conclusion

Summary

- Reproducibility is a concrete and hard problem. Nix helps.
- Nix expressions have shortcomings. We started the Nickel language to overcome them.
- There is a design space for alternative type systems. Gradual typing and first-class contracts is an exciting combo explored in Nickel.

CONFLANG21

Configuration languages are a worthy area of research.

The 1st Workshop on Configuration Languages

Website https://2021.splashcon.org/home/conflang-2021

Deadline Friday 6 August 2021

Duration 1 day

Event October 2021, at SPLASH 2021

The end

Nickel https://github.com/tweag/nickel/

Nix https://nixos.org/

Tweag's blog https://www.tweag.io/blog

Contact

- yann.hamdaoui@tweag.io
- hello@tweag.io