# Package Managers, through a lens or two

Jake Trevor

## If you want to follow along

Slides are here:

## Package managers are Ubiquitous

How many can you think of?

## Package managers are Ubiquitous

Some are familiar:

- utilities from the system PM (apt, brew, yum … )
- language specific dependencies (cargo, npm, pip, etc.)

Some are a little more exotic:

- ready-made actions in your CI (github actions)
- docker images on GHCR
- games on your iPad's app store
- minecraft mods via a mod manager (other games available)
- music from spotify

## Hygiene factors

- Despite how ubiquitous PMs are, no one spends any time thinking about them
- … because if you are thinking about your package manager, something has gone wrong.
- A good package manager is a *hygiene factor* - you notice it only by its absence

> No one notices a clean bathroom

**Similarly, no one ever wants to think about `pip`.**

## What is a package manager?

There are three characteristic problems package managers solve:

1. **Install** packages
2. Handle **dependencies**
3. (or perhaps, 2.a) Manage **versions**

Almost no one gets #1 wrong

But the rest is a lottery!

## What is a good package manager?

How should a well behaved package manager behave?

## Idea: Lenses

## Lenses

A lens $\mathcal{L}\,\Gamma\,\Delta$ on two types $\Gamma, \Delta$ is a pair of functions

$$\mathbf{Get}\ (\downarrow) : \Gamma \to \Delta \qquad \mathbf{Put}\ (\uparrow) : \Delta \times \Gamma \to \Gamma$$

$\Gamma$ is the source type - it represents '*the truth*'

$\Delta$ is the target type - it represents a *view* of the truth

## Lens Laws:

A lens is *well behaved* if it obeys the lens laws:

$$\forall\ (i : \Gamma)\ (x : \Delta),$$

$$\begin{array}{ccc} \textbf{put-get} & \qquad\qquad & \textbf{get-put} \\ \downarrow (\uparrow x\ i) = x & & \uparrow (\downarrow i)\ i = i \end{array}$$

and *very well behaved* if it is idempotent:

$$\begin{array}{c} \textbf{put-put} \\ \uparrow x\ (\uparrow x\ i) =\ \uparrow x\ i \end{array}$$

## Lenses?

Morally, this seems like it might be a good model ...

- We have some notion of *the truth* $(\Gamma)$ (the remote package repository)
- We want a *view of the truth* $(\Delta)$ (our local installation of packages)

but the devil is in the details

## Package lenses:

Let's say that we have:

- a type $P$ of package names
- and for each package $p : P$, a body of type $B$

A package manager is then a collection of lenses.

## Package lenses:

The source type is a map of names to bodies:

$$\Gamma = P \to B$$

For each subset of package names $P' \subseteq P$, we have a target type, which is a map from names to bodies:

$$\Delta_{P'} = P' \to B$$

## Package lenses:

In lean, we set this up using the *characteristic predicate* of the subset:

```
-- given h : P -> Pro

type P' = { p : P // h p }

type P' = Subtype h
```

The *package manager* itself can be understood as a function from the choice of packages to the corresponding lens:

$$\mathrm{PLens} : (h : P \to \mathrm{Prop}) \to \mathcal{L}\,\Gamma\,\Delta_{P'}$$

## Package lenses:

In this formalism, we have a pleasing set of subtypes: $P' <: P$ and $\Gamma <: \Delta_{P'}$

So **Get** is just a cast from $\Gamma$ to $\Delta_{P'}$.

```
get : (P -> Prop) -> (Subtype h -> Prop)
get := fun γ p => γ p
```

## Package lenses:

**Put** *overwrites* $\Gamma_P$ with values from $\Delta_{P'}$ whenever they exist:

```
put : ((Subtype h -> Prop) × (P -> Prop)) -> (P -> Prop)
put := fun (δ, γ) p =>
    if h p
    then δ p
    else γ p
```

## Package lenses

- Models the intuitive behaviour of installing packages
- All the lenses are *very well behaved*
  → I have lean proofs to back this up

## Dependencies

## The Problem: Dependency war

- Let's say that I use package $\mathrm{B}$
- and $\mathrm{B}$ depends on $\mathrm{A}$ version 1.
- Let's say I also want to use $\mathrm{C}$
- which depends on $\mathrm{A}$ version 2.
- Some combinations of packages are *impossible*
- How should a well behaved package manager deal with this?

## Dependency war

## Dependencies: Internal

- Often, a package *completely encapsulates* its dependencies.
- In this case, the dependencies are **internal** to the package.

→ We can use the package without knowing about its dependencies

## Dependencies: External

Packages can be *leaky*

- What if types from a dependency appear in the public interface of the package?
- Then the user of the package is *exposed* to the dependencies.
- These are called **external** dependencies

## To be clear:

- External dependencies make solving dependency war very hard.

**My view:**

- Dependency war between external dependencies in packages is *not* a technical problem.
- It's fundamental to the implementation of the packages.

Some sets of packages are *incompatible.*
Nothing a package manager can do will solve this.

## Aside: Can we fix it?

If you allow importing multiple versions of the same package, e.g. by naming them differently …

then you can expose both interfaces without a namespace collision

but doing so is contrary to the *spirit* of versioning

→ Versions provide different ways of doing the same thing
∴ it's weird to use two versions of the same thing

**However**, dependency war between internal dependencies should not be an issue.

**Solution: Dependency Locality**

## Local Dependencies:

The key idea is that we can load dependencies in a *localised* way:

1. When we load a dependency, it must not have side effects on some global environment
2. To support this, packages need to support *dependency injection*

Python does not have this facility; packages are referred to by name, as objects existing on the path

This is not a feature of the package manager, but of the underlying language/system

It's not pip's fault

**Pip sucks because python sucks**

## Module systems:

There's an extensive body of work on module systems, such as:

- **MixML** for ML (Rossberg & Dreyer)
- **backpack** for haskell (Kilpatrick & al.)
- $\Pi$ - a language agnostic module calculus; unpublished (Florisson & Mycroft)

These are much more complicated than I have time to go into here, and also a little out of scope.

## Versions

## Solution 1: new version, new name

- e.g. `pkg-A-v1` and `pkg-A-v2` are different packages
- This works very nicely with the theory we presented!
- It's also completely useless
    - in that it fails to model the actual problem
- *Dependency war* is just *namespace collision*
- If we really want to think about versions, we need to treat them as more than just 'new names'

## How should versions work?

- Versioning software is a solved problem
  → Just do what git does!
- What are the semantics of git?

## VCS semantics

We have:

- $S$ - a type of the *items* being versioned
- $H{<}S{>}$ - the type of histories
- $\mathrm{Hash}$ - an associated type of *hashes*

## VCS semantics

We don't care about the content of the types
so long as we can define the following two functions:

$$\begin{aligned} \mathtt{Commit} &: H{<}S{>} \times \mathrm{Hash} \times S \to H{<}S{>} \times \mathrm{Hash} \\ \mathtt{Checkout} &: H{<}S{>} \times \mathrm{Hash} \to S \end{aligned}$$

This might look familiar ...

$$\Gamma = H{<}S{>} \times \mathrm{Hash} \qquad \Delta = S$$
$$\mathtt{Get} = \mathtt{Checkout} \qquad \mathtt{Put} = \mathtt{Commit}$$

There's an infinite family of version control lenses (VCLs),
constructed by the function:

$$\mathrm{VCL} : (S : \mathrm{Type}) \to \mathcal{L}\ (H{<}S{>} \times \mathrm{Hash})\ S$$

## VCS: Is it a lens?

- If I checkout what I just committed, I should get the same thing back:

$$\mathbf{Checkout}\ (\mathbf{Commit}\ x\, i) = x$$

- If I check something out, and try to commit it, nothing will happen

$\rightarrow$ since Git will not (by default) allow empty commits.

$$\mathbf{Commit}\ (\mathbf{Checkout}\ i)\, i = i$$

- By the same fact, committing the same thing twice does nothing

$$\mathbf{Commit}\ x\, (\mathbf{Commit}\ x\, i) = \mathbf{Commit}\ x\, i$$

## So, in fact, this kind of abstract VCS is a lens, and it's *very well behaved!*

- Formalising this turned out to be hard
- I have a (very ugly) lean definition for VCLs
- But the proofs are not finished

## What do we want, morally?

- Let's say that we have a *history of bodies* for each package
- For a package $p : P$, let $V_p$ be the set of its versions

$\to$ i.e. hashes in its history

- Then for a set of packages $P'$, a *choice of versions* $V_{P'}$ is an object with type

$$V_{P'} = (p : P') \to V_p$$

## What do we want, morally?

Our source type should be a *map of histories* of package bodies:

$$\Gamma = P \to H{<}B{>}$$

As before, for a choice of packages $P'$ versions $V_{P'}$, we have a target type - a map of names to bodies:

$$\Delta_{V_{P'}} = P' \to B$$

We want a family of lenses, constructed by a function:

$$\text{mkPkgMan} = P' \times V_{P'} \to \mathcal{L}\left(P \to H{<}B{>}\right)\left(P' \to B\right)$$

## What do we have?

We can construct a package lens from names to histories

$$\mathrm{PLens\,ref} : \mathcal{L}\ (P \to H{<}B{>})\ (\mathrm{Subtype\,ref} \to H{<}B{>})$$

We can construct a VCL on those histories:

$$VCL : \mathcal{L}\ (H{<}B{>} \times \mathrm{Hash})\ B$$

Here are two other lenses:

$$ID : \mathcal{L}\ A\ A$$

$$\mathrm{Concentrate} : \mathcal{L}\ \left((B \to C) \times (B \to D)\right)\ (B \to (C \times D))$$

These are both vwb. (proofs in lean)

## Constructions on lenses:

Lenses can be tensored (parallel composition):

$$\_ \otimes \_ : \mathcal{L}\ A\ B \to \mathcal{L}\ C\ D \to \mathcal{L}\ (A \times C)\ (B \times D)$$

Composed (sequential composition):

$$\_ \circ \_ : \mathcal{L}\ A\ B \to \mathcal{L}\ B\ C \to \mathcal{L}\ A\ C$$

We can split a lens over a function in the second argument:

$$\_ \,\mathrm{split}\, \_ : \mathcal{L}\ A\ (B \to C) \to \mathcal{L}\ C\ D \to \mathcal{L}\ A\ (B \to D)$$

These constructions are *behaviour preseving* - their output is vwb. when the input lenses are.

## What we get: Versioned Package Lenses

$$\begin{aligned}\mathrm{VPL}\,\mathrm{ref} = ((&\mathrm{PLens}\,\mathrm{ref} \otimes \mathrm{ID}) \\ &\qquad\quad \circ \mathrm{Concentrate}) \\ &\mathrm{split}\,(\mathrm{VCL}) \\ : (P \to \mathrm{Prop}) \to \mathcal{L}\; & \\ &((P \to H{<}B{>}) \times V_{P'}) \\ &(P' \to B)\end{aligned}$$

Since this is formed only by vwb. lenses and behaviour preserving constructions, the result is also vwb.

## What we get isn't quite what we wanted

$$\mathcal{L}\left(\left(P \to H{<}B{>}\right) \times V_{P'}\right)\left(P' \to B\right)$$

The target type is correct

$$\Delta = \left(P' \to B\right)$$

but the source type is a little different:

$$\begin{array}{cc} \text{Wanted:} & \text{Got:} \\ \Gamma = P \to H{<}B{>} & \Gamma = \left(P \to H{<}B{>}\right) \times V_{P'} \end{array}$$

## The difference

The source type includes our choice of versions $V_{P'}$

Intuitively: **Put** can change our choice of versions

- From a certain point of view, this makes sense
  - The version you just pushed is not the one you checked out
- From another, it does not:
  - If versions can change with a push, why not the set of packages?
  - i.e. what's the story for 'pushing a new package'?

It may well be that a different approach might yield a more pleasing result

wherein versions are names are treated more similarly.

Food for thought

## The wrinkles

- Since the formalisation of VCL is not done, I don't have a verified proof that this is all OK just yet.
- At present, while 'tensoring' is morally right, it's not sufficient.
- We need a dependent tensor - where the second argument can depend on the first

## Take care!

Notice that for us, the package registry is a *map of histories*:

$$P \to H{<}B{>}$$

But Nix instead has a *history of maps*:

$$H{<}P \to B{>}$$

nixpkgs (the central repository) is a versioned map of names to recipes

Which is the right way around?

## My view: Nix does it wrong

With Nix' system, we refer to packages in a particular version of nixpkgs

**Problem 1**: bumping a version might not change your installation at all
→ i.e. if the packages were not changed in that commit

**Problem 2**: Versioned versions

## Versioned Versions

Nix allows you to refer to different nixpkgs versions
to allow you to refer to specific builds of a given package

But this can be cumbersome

∴ different versions of a package are often distributed as *different packages*

e.g. php82 and php83 are different packages.

php82 might exist in 2 commits

You can refer to *different versions of php82*

→ Ridiculous

## Conclusions

- We can model package managers with lenses
    - Lots of the nuance is nicely captured
- Particularly, simple unversioned package managers
- For good dependency behaviour, the underlying system needs a good module system
    - Even if we get everything else right, we can still get trapped by non-local dependencies
- This formalism has some weaknesses, space for development
    - A more equal treatment of versions and package names would be nice

## Future work?

- What does proper behaviour for **shared dependencies** look like?
  → What's the next best thing after full locality?
  → What about version constraints?
- **Extra features**
  - e.g. declarative/imperative interfaces, reversibility and uninstallation
    → some of this is already captured by the models I've already presented
- **Security**
  → Push should fail sometimes - is that worth modelling?

Some references:

[1] Florisson & Mycroft, *Towards a Theory of Packages unpublished draft* (2016)
[2] Rossberg & Dreyer, *Mixin' Up the ML Module System* (2013)
[3] Kilpatrick & al, *Backpack: Retrofitting Haskell with Interfaces* (2014)