Lenses

Functional Programming II

Lukas Pietzschmann
lukas.pietzschmann@uni-ulm.de

Institute of Software Engineering and Programming Languages
Ulm University

May 13th, 2024


This image was generated by AI (DALLE 3)

Agenda

Learning Objectives

Why do we need lenses?

I

Understand where the idea of lenses come from, and how one could have come up with them.

What else is there?

III

Know of other lens-like abstractions, why we presumably need them, and how they differ.

How can I use them?

II

Know the basic functions and operators and know how to discover new ones.

WTF are those types?

IV

Understand the ins and outs of the lens package and every type.

1. What

1 What are lenses

```haskell
type Lens s t a b = forall f. Functor f ⇒ (a → f b) → s → f t
```

What is the purpose of a lens, according to the types above?

- **A:** A package for creating visualizations
- **B:** A tool for handling nested ADTs
- **C:** A framework for building UIs
- **D:** A package for simulating optical lenses

1 What are lenses

```haskell
type Lens s t a b = forall f. Functor f ⇒ (a → f b) → s → f t
```

What is the purpose of a lens, according to the types above?

A: A package for creating visualizations

B: A tool for handling nested ADTs

C: A framework for building UIs

D: A package for simulating optical lenses

1 WHAT ARE LENSES

```
type Lens s t a b = forall f. Functor f ⇒ (a → f b) → s → f t
```

> *In Haskell, types provide a pretty good explanation of what a function does. Good luck deciphering lens types.*
>
> Roman Cheplyaka

A: A package for creating visualizations

B: A tool for handling nested ADTs

C: A framework for building UIs

D: A package for simulating optical lenses

1 What are lenses

Well, "lens" is also a package ... Here are some random functions and operators from that package:

`view`	`_1`	`allOf`
`set`	`^.`	`anyOf`
`over`	`^?!`	`concatOf`

We'll shortly see what they do and how we can use them.

2. Why

2.1 WHY DO WE NEED THEM

Imagine you want to parse configuration files in Haskell. To model them, you come up with the following ADTs:

```
data File = File {
  name    :: String,
  entries :: [Entry]
}
data Entry = Entry {
  key   :: String,
  value :: Value
}
data Value = Value {
  curr :: String,
  def  :: String
}
```

Why do we need them

Let's say we parsed a file into the following configuration:

```
config = File "~/.config/nvim/init.lua" [
    Entry "expandtab" (Value "" "true"),
    Entry "cmdheight" (Value "0" "1"),
    Entry "textwidth" (Value "88" "")
  ]
```

Cool, isn't it. Now we want to work with this representation.

2.3 Why do we need them

```haskell
data File = File {
  name    :: String,
  entries :: [Entry]
}
data Entry = Entry {
  key   :: String,
  value :: Value
}
data Value = Value {
  curr :: String,
  def  :: String
}
```

```haskell
getEntry :: String → File → Entry
getEntry k = head . filter ((==) k . key) . entries

getCurrentValue :: Entry → String
getCurrentValue = curr . value

setCurrentValue :: String → Entry → Entry
setCurrentValue newValue entry = entry {
    value = (value entry) {
        curr = newValue
    }
}
```

Oof, this sucks. And it get's even worse the deeper the ADT gets!

Let's reinvent the lens

```haskell
data File = File {
  name    :: String,
  entries :: [Entry]
}
data Entry = Entry {
  key   :: String,
  value :: Value
}
data Value = Value {
  curr :: String,
  def  :: String
}
```

Let's see, if we can improve this by adding some modifier functions:

```haskell
modifyCurrentValue :: (String → String) → Value → Value
modifyCurrentValue f value = value {
    curr = f $ curr value
}

modifyEntriesValue :: (Value → Value) → Entry → Entry
modifyEntriesValue f entry = entry {
    value = f $ value entry
}

modifyEntriesCurrentValue :: (String → String) → Entry → Entry
modifyEntriesCurrentValue = modifyEntriesValue . modifyCurrentValue
```

LET'S REINVENT THE LENS

```haskell
data File = File {
  name    :: String,
  entries :: [Entry]
}
data Entry = Entry {
  key   :: String,
  value :: Value
}
data Value = Value {
  curr :: String,
  def  :: String
}
```

We can use our modify-functions to implement a setter:

```haskell
setCurrentValue' :: String → Entry → Entry
setCurrentValue' = modifyEntriesCurrentValue . const
```

The getter is still fine:

```haskell
getCurrentValue' :: Entry → String
getCurrentValue' = def . value
```

Let's reinvent the lens

Now, we can build our lens abstraction:

```
data Lens s a = Lens {
    get :: s → a,
    modify :: (a → a) → s → s
}
```

We need to reimplement the function composition:

```
compose :: Lens a b → Lens b c → Lens a c
compose (Lens g m) (Lens g' m') = Lens {
    get = g' . g,
    modify = m . m'
}
```

For easier handling, we also define `set` as a little helper:

```
set :: Lens s a → a → s → s
set (Lens _ modify) = modify . const
```

Let's reinvent the lens

Finally, we can build lenses for our ADTs:

```haskell
data File = File {
  name    :: String,
  entries :: [Entry]
}
data Entry = Entry {
  key   :: String,
  value :: Value
}
data Value = Value {
  curr :: String,
  def  :: String
}
data Lens s a = Lens {
  get :: s → a,
  modify ::
    (a→a) → s → s
}
```

```haskell
currentValueL :: Lens Value String
currentValueL = Lens {
    get = curr,
    modify = \f value → value { curr = f $ curr value }
}

entryValueL :: Lens Entry Value
entryValueL = Lens {
    get = value,
    modify = \f entry → entry { value = f $ value entry }
}

entryCurrentValueL :: Lens Entry String
entryCurrentValueL = entryValueL `compose` currentValueL
```

Let's reinvent the lens

Now we only have to plug our lens into `set`, `get`, or `modify`:

```
setCurrentValue'' :: String → Entry → Entry
setCurrentValue'' = set entryCurrentValueL

getCurrentValue'' :: Entry → String
getCurrentValue'' = get entryCurrentValueL
```

2.9 LET'S REINVENT THE LENS

Puh, that was kinda complicated. But again, think of how much less code you have to write:

```haskell
let f = _foo v
    b = _bar f
    z = _baz b in
v { _foo = f {
        _bar = b {
            _baz = z + 1
        } } }
```

versus

```haskell
v & foo . bar . baz +~ 1
```

We can now think "How can I traverse through this?" instead of "How do I un- and repack all of this?".

2.10 LET'S REINVENT THE LENS

Our solution looks more flexible than what we had before. But there are still some problems:

- Still feels a bit clunky and boilerplate-heavy
- We always have to create `Lens` values
- No support for polymorphic updates

It's definitely not impossible to overcome these limitations, but we'll skip this for now.

Let's Reinvent the Lens

Polymorphic Update

```
data Pair a b = Pair {e1 :: a, e2 :: b}

p :: Pair Int String
p = Pair 420 "is fun"

p { e1 = "FP" } ▸ Pair { e1 = "FP", e2 = "is fun" }
```

- Notice that the type has changed from `Pair Int String` to `Pair String String`. This is what we call *polymorphic update*.

…ut there

but we'll

2.11 What are lenses

Revisited

Lenses are:

- A way to *focus* on a part of a data structure

Or more precisely:

- Just another abstraction
- Functional references
- Getters and Setters
- Highly composable and flexible
  - “The Power is in the Dot” Edward Kmett

A LITTLE HISTORY LESSON

[Kme20]

- Luke Palmer creates a pattern he calls *Accessors* to ease stateful programming in Haskell [Pal07b]. He uses C's preprocessor to generate `readVal` and `writeVal` functions.[1]
- Palmer generalizes his Accessors into something more like today's lenses. [Pal07a]
- Twan van Laarhoven comes up with a novel way to express lenses using the **`Functor`** class [Laa09]. We call them *van Laarhoven lenses*.

[1] In another blog post he then swaps out the preprocessor in favour of Template Haskell.

A little history lesson

- Russell O'Connor realises van Laarhoven lenses have always supported polymorphic updates. [OCo12]
- Edward Kmett realises that you can put laws on the notion of polymorphic updates. [Kme12]
- Kmett pushed the first commit to the lens repository on GitHub

3. How

3.1 A little Overview

Lenses basically provide two kinds of operations:

- `view :: Lens' s a → s → a`
- `set :: Lens' s a → a → s → s`

To use them, we need the actual lens. It determines what part of the structure we want to focus on.

- `_1 :: Lens' (a,b) a`
- `_2 :: Lens' (a,b) b`

With all that in place, we can now combine the operation with a lens (or a combination of lenses) and data:

- `set _2 "cool" ("FP is", "")` ▸ `("FP is", "cool")`
- `view _1 ("hi", "there")` ▸ `"hi"`

3.2.1 Lens Laws

Like with functors, applicatives, and monads, lenses *should* follow some rules:

1. Get-Put
2. Put-Get

🐣 Put-Put

We'll look at them in a bit more detail.

Lens Laws

If you modify something by changing its subpart to exactly what it was before, nothing should happen.

```
set entryValueL (get entryValueL entry) entry == entry
```

- The lens should not modify the value or structure by itself.

Lens Laws

If you modify something by inserting a particular subpart and then view the result, you'll get back exactly that subpart.

```
get entryValueL (set entryValueL v entry) == v
```

- Setting values should be independent of any previous state.

Lens Laws

If you modify something by inserting a particular subpart `a`, and then modify it again inserting a different subpart `b`, it's exactly as if you only did the second insertion.

```
set entryValueL v2 (set entryValueL v1 entry) == set ↩
    entryValueL v2 entry = 1
```

- Previous updates should not leave any traces.

Do I really have to follow them?

- Yes, you should! Otherwise your lenses might behave weird.
- And weird unpredictable things are for OOP 😉

- But, we can get around them
- In fact, we can get around the whole process of creating a lens by hand
- You remember Template-Haskell, do you?

3.2.6 Do I really have to follow them?

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens

data File  = File  {_name :: String, _entries :: [Entry]}
data Entry = Entry {_key  :: String, _value   :: Value  }
data Value = Value {_curr :: String, _def     :: String }

makeLenses ''File
makeLenses ''Entry
makeLenses ''Value
```

3.3.1 The lens Package

- Until now, we have only used `view` and `set`
- But there are actually a lot more functions and operators
- I mean a looooooooooooooooooot; easily over 100

- Let’s try to find a pattern in their names

3.3.2 The lens Package

Operators beginning with `^` behave like `view` functions:

```
Value "c" "d" ^. def ▸ "d"
(1,2) ^.. both ▸ [1,2]
Right 42 ^? _Left ▸ Nothing
```

Operators ending in `~` behave like `set` functions:

```
(_2 .~ 3) (0, 0) ▸ (0,3)
(_2 +~ 3) (0, 39) ▸ (0,42)
(_1 %~ (+1)) (3,2) ▸ (4,2)
```

Writing `lens .~ value $ adt` every time is not very nice. But as always, there's a special operator to our rescue: `& :: a → (a → b) → b`.

The lens Package

```
config = File "~/.config/nvim/init.lua" [
  Entry "expandtab" (Value "" "true"),
  Entry "cmdheight" (Value "0" "1"),
  Entry "textwidth" (Value "88" "")
  ]
```

With this knowledge aquired, we can finally write concise Haskell-code:

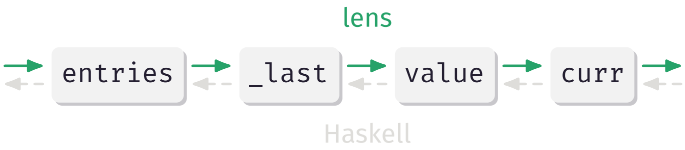
```
(6, 2) & both *~ 7 ▸ (42, 14)

lens = entries . _last . value . curr
val = config ^?! lens ▸ "88"
config & lens .~ val ++ "0" ▸ curr = "880" inside config
over lens (++"0") config ▸ curr = "880" inside config

(0, "upd.") & _1 .~ "poly." ▸ ("poly.", "upd.")
```

The lens Package

You might have notices that lenses compose backwards:



This makes it weird for FP-enjoyers, but intuitive for OOP-weirdos.
The same applies for all kinds of operators:

lens	Haskell
5 & (+1)	(+1) $ 5
Just 5 <&> (+1)	(+1) <$> (**Just** 5)

You might have notices that lenses compose backwards:

lens

> *Backward composition of lenses. It's a minor issue, and I wouldn't mention it if it wasn't a great demonstration of how lens goes against the conventions of Haskell.*
>
> Roman Cheplyaka

eirdos.

lens	Haskell
`5 & (+1)`	`(+1) $ 5`
`Just 5 <&> (+1)`	`(+1) <$> (Just 5)`

The lens Package

Writing a *Getter* is really easy. We can simply promote any *function* or *value* to a Getter.

- `to` builds a Getter from any function

```
("Hello", "FP2") ^. to snd ▸ "FP2"
```

- `like` always returns a constant value

```
("Hello", "FP2") ^. like 42 ▸ 42
```

The lens Package

Writing a *Setter* is only slightly more complicated, as we don't set the value directly, but apply a function on the focused part.

- `setting` receives a function, that applies another function to the correct value inside a structure

```
(4,1) & setting (\f (x,y) → (x,f y)) .~ 2 ▸ (4,2)
```

- `sets` is in theory a bit more flexible, but that's out of scope for today

```
(4,1) & sets (\f (x,y) → (x,f y)) .~ 2 ▸ (4,2)
```

The lens Package

Having a separate Getter and Setter is not always desirable. Now, we want to create our own lens that we can use as both Getter and Setter. This time, `makeLenses` doesn't count!

- We can use `lens` to combine a viewing and setting function

```
g = snd
s = (\(a,_) b → (a,b))
_2 = lens g s
```

- You can also simply write a custom function with the type `l :: forall f. Functor f ⇒ (a → f b) → s → f t` that satisfies all three lens laws. Good luck! We'll try it anyway.

The lens Package


```
type Lens s t a b = forall f. Functor f ⇒ (a → f b) → s → f t
type Lens' s a = Lens s s a a
```

- `a` — The inner type we're interested in
- `s` — The type of the whole structure

```
lens :: Functor f ⇒ (s→a) → (s→a→s) → (a→f a) → s → f s
lens get set f s = ...
```

- We need to get from `s → a` and `s → a → s` to `f s`
- We can get an `a` from our getter: `get s`
- With `a` and `f` we can make an `f a`: `f $ get s`

```
lens :: Functor f ⇒ (s→a) → (s→a→s) → (a→f a) → s → f s
lens get set f s = set s <$> f (get s)
```

- We need to get from `s → a` and `s → a → s` to `f s`
- We can get an `a` from our getter: `get s`
- With `a` and `f` we can make an `f a`: `f $ get s`
- Now, to get an `f s`, we an simply use

```
fmap :: Functor f ⇒ (a → b) → f a → f b
```

`(a → b)` ← `set s`

`f a` ← `f $ get s`

4. More Goodies

4.1 Virtual lenses

A Getter does not always have to be backed by an actual structure. Theoretically, it can return *anything*:

```
get virtualProp(): number {
    return 42
}
```

We can easily achieve this behavior with lenses, too:

```
virtualProp = like 42
(0,0) ^. virtualProp ▸ 42
```

4.2.1 Prisms

So far, we only looked at product types. But what about sum types? Prisms to the rescue!

```
data CanteenMeal = MainCourse String CanteenMeal
                 | Desert String

meal1 = MainCourse "Sattmacher" (Desert "Pudding")
meal2 = Desert "Yogurt"

meal1 ^? _MainCourse . _2 . _Dessert ▸ Just "Pudding"
meal2 ^? _MainCourse . _2 . _Dessert ▸ Nothing

meal1 & _MainCourse . _2 . _Dessert .~ "Yogurt"
▸ Desert "Yogurt" inside meal1
```

4.2.2 Prisms

- We already used a prism: remember `_last`?
- We can usually use them like a normal lens (there's just a little `Maybe` in the way)

```
case meal1 of
    MainCouse _ (Dessert d) → MainCourse {
        dessert = Dessert "Yogurt" }
    _ → meal1
```

versus

```
meal1 & _MainCourse . _2 . _Dessert .~ "Yogurt"
```

4.3.1 Traversals

Wouldn't it be nice to have a lens that focuses on a specific element of a traversable container? Let's start with every element:

```
["Hello", "there"] ^. traverse ▸ "Hellothere"
```

Huh?! What's that? I would've expected `["Hello", "there"]`. When viewing the result of `traverse`, it gets shoved through `mappend` first. That's why you typically `^..`.

```
[1..5] ^.. traverse ▸ [1,2,3,4,5]
[(1,2),(3,4)] ^.. traverse . _2 ▸ [2,4]

[1..5] & traverse +~ 1 ▸ [2,3,4,5,6]
```

4.3.2 Traversals

As promised, here's how we can focus on a specific element of a traversable:

```
[1..5] ^.. ix 1 ▸ [2]
[1..5] ^.. ix 5 ▸ []
```

Returning an empty list on failure does not seem very nice. Let's use the prism-view-operator to get a `Maybe`:

```
[1..5] ^? ix 1 ▸ Just 2
[1..5] ^? ix 5 ▸ Nothing
```

4.4.1 Isos

Here's a very short summary:

- An `Iso` is a connection between two types that are equivalent in every way
- Isos should follow the following laws:
```
forward . backward = id
backward . forward = id
```
- We can write our own `Iso` by providing a forward and backward mapping

4.4.2 Isos

```haskell
maybeToEither = maybe (Left ()) Right
eitherToMaybe = either (const Nothing) Just

someIso :: Iso' (Maybe a) (Either () a)
someIso = iso maybeToEither eitherToMaybe

Just "hi" ^. someIso ▸ Right "hi"
Left "ho" ^. from someIso ▸ Nothing
```

5. Summary

5.1 Summary

Traversals

- Focus on multiple parts (also zero) of a data structure
- `^..` returns list of the focused parts

↗ ↖

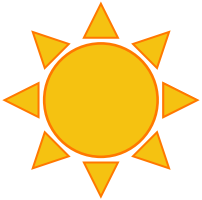
Lens

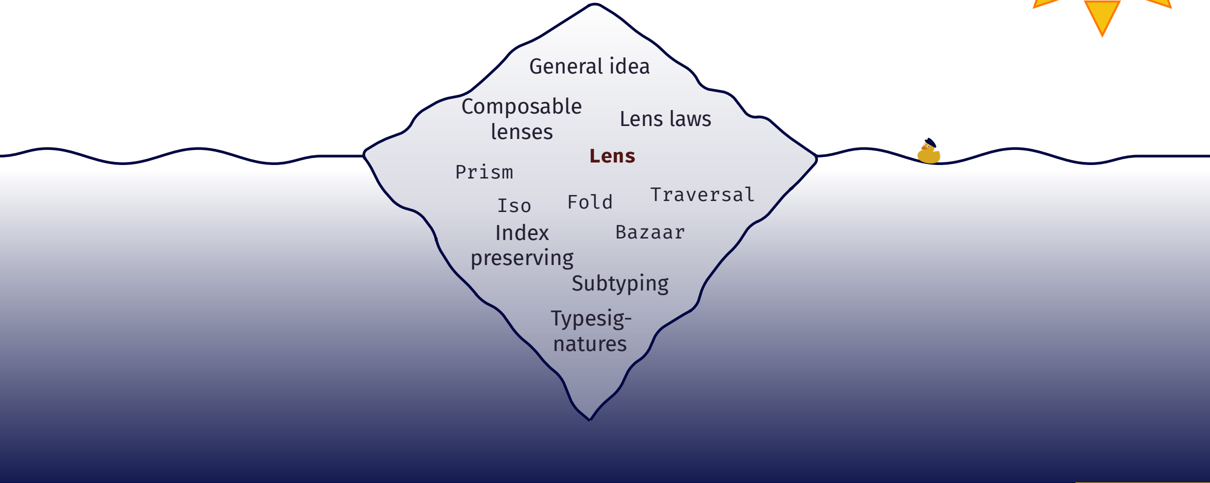
- Focus on a single part of a data structure
- `^.` returns the focused part directly

Prism

- Focus on a single part that may not exist
- `^?` returns the focused part inside a **`Maybe`**

5.2 AND SO MUCH MORE





6. References

6.1 READING SUGGESTIONS (I)

[Abr18] Joseph Abrahamson. *A Little Lens Starter Tutorial*. 2018. URL: https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial (visited on 03/05/2024).

[Bha13] Aditya Bhargava. *Lenses In Pictures*. 2013. URL: https://www.adit.io/posts/2013-07-22-lenses-in-pictures.html (visited on 04/25/2024).

[Laa09] Twan van Laarhoven. *CPS based functional references*. 2009. URL: https://www.twanvl.nl/blog/haskell/cps-functional-references (visited on 04/18/2024).

6.1 READING SUGGESTIONS (II)

[OCo12] Russell O'Connor. *Polymorphic Update with van Laarhoven Lenses*. 2012. URL: https://r6.ca/blog/20120623T104901Z.html (visited on 04/18/2024).

[Rom19] Veronika Romashkina. *Write yourself a lens*. 2019. URL: https://vrom911.github.io/blog/write-yourself-a-lens.

6.2 References (I)

[Abr18] Joseph Abrahamson. *A Little Lens Starter Tutorial*. 2018. URL: https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial (visited on 03/05/2024).

[Bha13] Aditya Bhargava. *Lenses In Pictures*. 2013. URL: https://www.adit.io/posts/2013-07-22-lenses-in-pictures.html (visited on 04/25/2024).

[Che14] Roman Cheplyaka. *Lens is unidiomatic Haskell*. 2014. URL: https://ro-che.info/articles/2014-04-24-lens-unidiomatic.html (visited on 04/09/2024).

6.2 References (II)

[GO09] Jeremy Gibbons and Bruno C d S Oliveira. "The essence of the iterator pattern". In: *Journal of functional programming* 19.3-4 (2009), pp. 377–402.

[Kme12] Edward Kmett. *Mirrored Lenses*. 2012. URL: https://web.archive.org/web/20240301015449/https://comonad.com/reader/2012/mirrored-lenses/ (visited on 03/01/2024).

[Kme20] Edward Kmett. *History of Lenses*. 2020. URL: https://github.com/ekmett/lens/wiki/History-of-Lenses (visited on 04/18/2024).

[Laa09] Twan van Laarhoven. *CPS based functional references*. 2009. URL: https://www.twanvl.nl/blog/haskell/cps-functional-references (visited on 04/18/2024).

6.2 References (III)

[OCo12] Russell O'Connor. *Polymorphic Update with van Laarhoven Lenses.* 2012. URL: https://r6.ca/blog/20120623T104901Z.html (visited on 04/18/2024).

[Pal07a] Luke Palmer. *Haskell State Accessors (second attempt: Composability).* 2007. URL: https://web.archive.org/web/20120303223802/https://lukepalmer.wordpress.com/2007/08/05/haskell-state-accessors-second-attempt-composability/ (visited on 03/03/2012).

6.2 References (IV)

[Pal07b] Luke Palmer. *Making Haskell nicer for grame programming*. 2007. URL: https://web.archive.org/web/20220222032352/https://lukepalmer.wordpress.com/2007/07/26/making-haskell-nicer-for-game-programming/ (visited on 02/22/2022).

[Rom19] Veronika Romashkina. *Write yourself a lens*. 2019. URL: https://vrom911.github.io/blog/write-yourself-a-lens.

6.2 References (V)

[Wik23] Wikibooks. *Haskell/Lenses and functional references*. 2023. URL: https://en.wikibooks.org/w/index.php?title=Haskell/Lenses_and_functional_references&oldid=4342240 (visited on 03/05/2024).

Lukas Pietzschmann

Ulm, May 13th, 2024

lukas.pietzschmann@uni-ulm.de