@AndrzejWasowski
Andrzej Wąsowski

Advanced Programming

Lenses: Functional updates to complex values

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

Mentimeter

Motivating example in Lecture.scala

Lenses

Summary

- Typically **easy to navigate** purely to values (get) but difficult to create deep nested assignments
- The **more complex** the structure, **harder to modify** it purely
  - Very hard for XML, JSon, and YAML schema
  - Foster et al. use XML as an example
  - The principle extends to any trees (e.g. abstract syntax trees of program code)

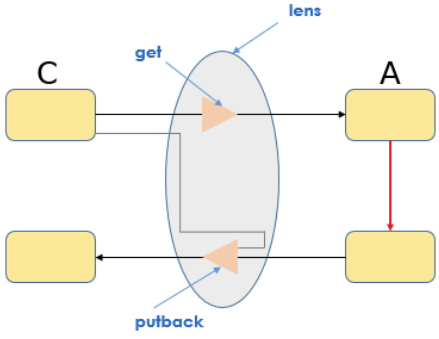
Lenses are a uniform architecture and an algebra of combinators to systematically and simultanously create set and get functions for complex structures.

- Specify the **get** and **set** (**put**) functions (expected)
- Lenses provide a way to **compose nested setters and getters**
- Guarantee that **algebraic laws capturing well behavedness** hold
- For a new lense you build, **test laws with PBT**

Lens

Definition

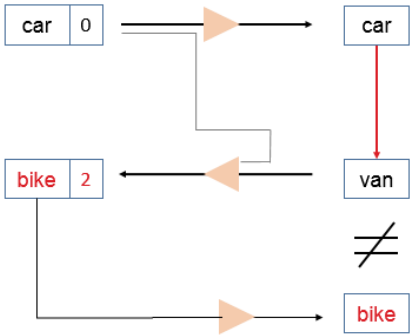
Def. A lense $\mathtt{l}$ from concrete (larger) representation (type) C to an abstract (smaller) representation (type) A comprises a partial function $\mathtt{l}\nearrow\colon \mathrm{A} \to \mathrm{C}$ (`get`) and a function $\mathtt{l}\searrow\colon \mathrm{A} \to \mathrm{C} \to \mathrm{C}$ (AKA `putback` / `put` / `set`)



- In Monocle (the library we use), lenses are (roughly) called optics
- We mostly look at three types of optics:
  - `Lens[C,A]` (total lens)
  - `Optional[C,A]` (a partial lense),
  - `Traversal[C,A]` (access elements in a collection)

Mentimeter

Put-Get Law



- Consider a lens that operates on vehicle-number records
- It extracts a view on the vehicle type (a usual getter/setter for a field)
- We extract value `car` and want to put back `van`
- When we get the value of vehicle type again we would like to get a `van`, not something else!

Put-Get Law

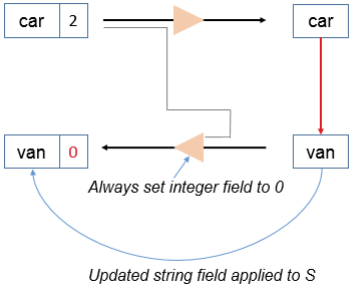
For a lense $\mathtt{l}$, each concrete value $\mathtt{c}$ and each abstract view $\mathtt{a}$ we got: $\mathtt{l} \nearrow (\mathtt{l} \searrow (\mathtt{a})(\mathtt{c})) = \mathtt{a}$

In Scala/Monocle: `l.get (l.set (a) (c)) == a`

Foster et al. formulate the law using an equality that makes sense for partial lenses. too.

Mentimeter

Get-Put Law



- Consider a lens that operates on vehicle-number records (as before)
- It extracts a view on the vehicle type (as before)
- On `put` it always sets the number to zero, regardless of what was there before
- This is a confusing side-effect for a setter!

Get-Put Law

For a lense `l` and each concrete value `c` we have: $\texttt{l} \searrow (\texttt{l} \nearrow (\texttt{c}))(\texttt{c}) = \texttt{c}$

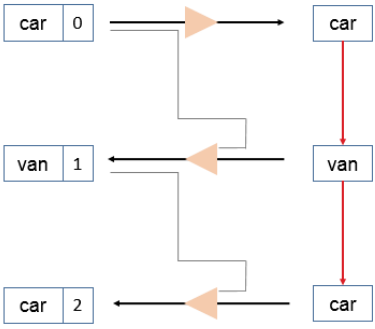
In Scala/Monocle: `l.set (l.get (c)) (c) == c`

Again, for partial lenses we only eforce the law if set/get do not fail.

Def. A lens satisfying Put-Get and Get-Put is called **well-behaved**.

Mentimeter

Put-Put Law



- Consider a lens that operates on vehicle-number records (as before)
- It extracts a view on the vehicle type (as before)
- This lens, has another problem, even though we put `car` second time, we obtain a different record than before.
- The putting of `van` is not completely anihilated!

Get-Put Law

For a lens $\mathtt{l}$, and all values $\mathtt{c}$, $\mathtt{a}$, and $\mathtt{a}'$ we have: $\mathtt{l} \searrow (\mathtt{a}')(\mathtt{l} \searrow (\mathtt{a})(\mathtt{c})) == \mathtt{l} \searrow (\mathtt{a}', \mathtt{c})$

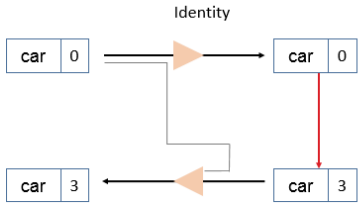
In Scala syntax: `l.set (a1) (l.set (a) (c)) == l.set (a1) (c)`

Def. A lens satisfying Put-Get, Get-Put, and Put-Put is called **very well-behaved**.

Mentimeter

Identity

An example Lens



- A lens that gets the entire object
- And updates the entire object
- **Question:** what is `identityLens[Int].get (42)`?
- **Question:** what is `identityLens[Int].set (42) (13)`?

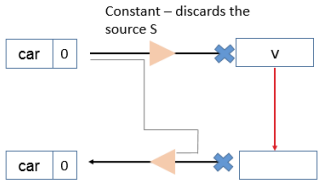
In Scala syntax:

```
def identityLens[A] = Lens[A,A] (c => c) (c => a => a)
```

Total, very well-behaved.

Constant

An example Lens



- A lens that always reads the same value
- And does not modify the concrete objects
- **Question:** what is `constLens (13).get (42)`?
- **Question:** what is `constLens (13).get (42).set (13)`?

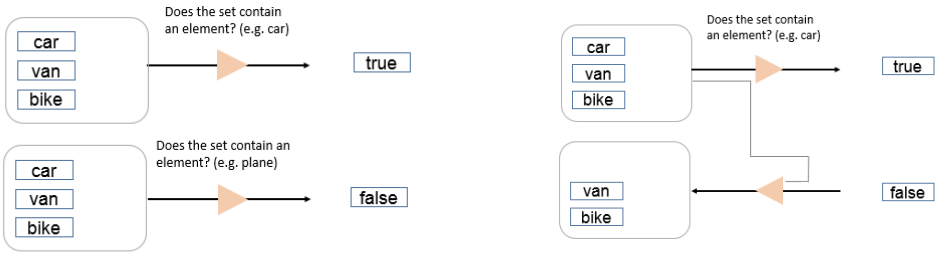
In Scala syntax:

```
def constLens[C,A] (default: A) =  Lens[C,A] (c => default) (_ => c => c)
```

Total, not well-behaved.

Set Membership (Contains)

An example Lens



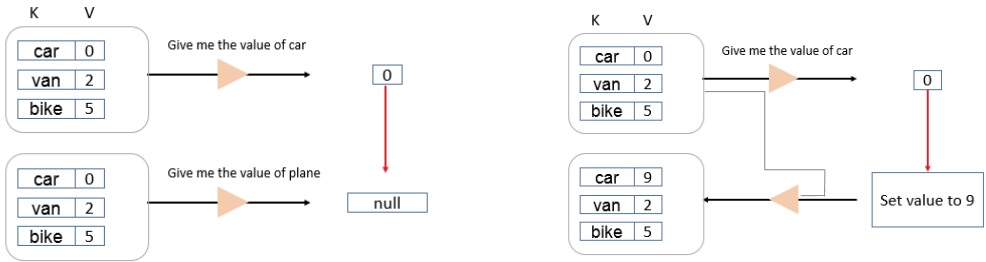
In Scala syntax:

```
def contains[T] (x: T) =
  Lens[Set[T],Boolean]
    (get = _.contains (x))
    (set = b => c =>if (b) c.incl (x) else c.excl (x))
```

Total, very well-behaved.

Index (in a map)

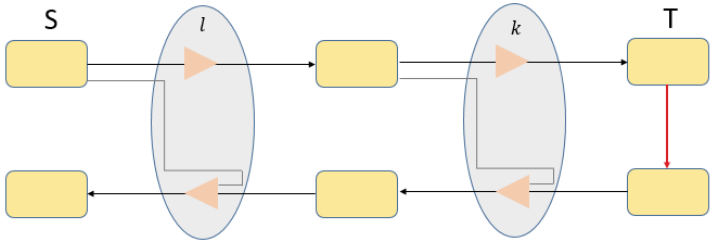
An example Lens



In Scala syntax:

```
1 def index[K,V] (k: K): Optional[Map[K,V],V] = {
2   def get (m: Map[K,V]): Option[V] = m.get (k)
3   def set (v: V) (m: Map[K,V]): Map[K,V] = m + (k->v)
4   Optional[Map[K,V],V] (get)  (set _)
5 }
```

Composing Lenses




```
1  def compose[S,A,T] (l: Lens[S,A]) (k: Lens[A,T]): Lens[S,T] = {
2    def get (s: S): T = k.get (l.get (s))
3    def set (t: T) (s: S): S =
4      l.set (k.set (t) (l.get (s))) (s)
5    Lens[S,T]  (get) (set _)
6  }
```

A composition of total lenses is total, a composition of well-behaved lenses is well-behaved

Lenses

Concluding Remarks

- There are many lens libraries for Scala (and other functional languages)
- AFAIK, the first implementation was in Haskell
- Monocle uses slightly different identifiers and types
- It also uses type classes (implicits), macros, and annotations to derive some lenses automatically
- All this we know so that you are now well equipped to read
  `https://www.optics.dev/Monocle/docs/optics/lens`

Credits: I used figures from presentations of Nate Foster in this lecture