Lenses

Lecture 110 of Advanced Programming

November 1, 2018

Andrzej Wasowski & Zhoulai Fu

IT University of Copenhagen

Case class

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
  left: Expr, right: Expr) extends Expr
```

Scala's way to allow pattern matching on objects without requiring a large amount of boilerplate.

How to update a field in a case class?

- - case class A(i : Int)
- - val v = A (5)
- How can I change “i” field to 6?
  - Old way: A(v.i+1)
  - New way: v.copy(i = 6)

What if the field is nested?

```
// imperative
a.b.c.d.e += 1

// functional
a.copy(
  b = a.b.copy(
    c = a.b.c.copy(
      d = a.b.c.d.copy(
        e = a.b.c.d.e +
          1))))
```

Natural reactions

- 1. FP programming sucks
- 2. We should do something
- Let us do something.

Lens

A Lens is an abstraction from functional programming which helps to deal with a problem of updating complex immutable nested objects.

- - case class A(i : Int)
- - val v = A (5)
- How can I change “i” field to 6?
  - Old way: A(v.i+1)
  - New way: v.copy(i = 6)
  - Lens automates, and generalizes the “New way”

Monocle Lens

- Monocle is a scala library
- - import monocle.Lens
- - If you want to change the “a” filed of the “cass class A(a:Int)”, do this:
  - Val mylense=Lense[A,Int](getter)(setter)
  - - getter: A=>Int
  - - setter: Int =>A =>A

Laws for Lens

From Monocle package homepage:

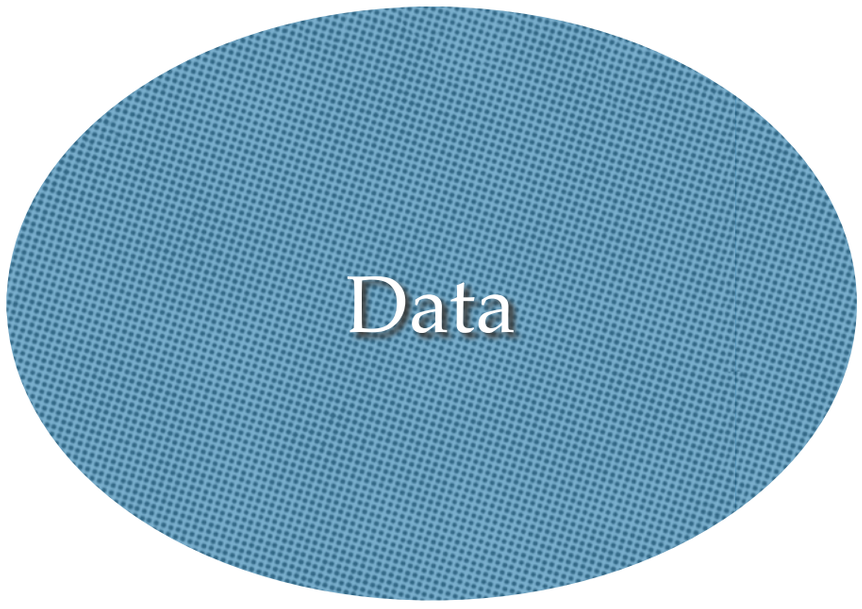
> A `Lens` must satisfy all properties defined in `LensLaws` from the `core` module. You can check the validity of your own `Lenses` using `LensTests` from the `law` module.

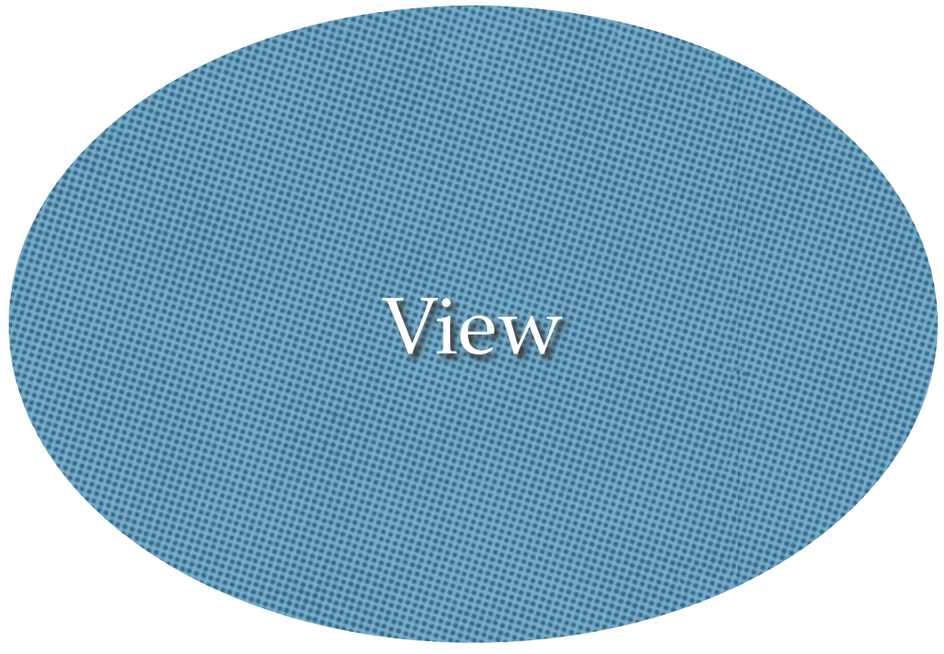
- - case class A(i:int)
- - l.set(l.get(a))(a) == a
- - l.get(l.set(i)(a)) == i

The bigger picture — The view-update problem

Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem

J. Foster, Michael Greenwald, Jonathan Moore, Benjamin Pierce, Alan Schmitt





View update problem:

Update on View must be correctly reflected on updates on Data

When f is a partial function, we write $f(a)\downarrow$ if f is defined on argument a and $f(a) = \bot$ otherwise. We write $f(a) \sqsubseteq b$ for $f(a) = \bot \vee f(a) = b$. We write $\mathsf{dom}(f)$ for $\{s \mid f(s)\downarrow\}$, the set of arguments on which f is defined. When $S \subseteq \mathcal{V}$, we write $f(S)$ for $\{r \mid s \in S \ \wedge \ f(s)\downarrow \ \wedge f(s) = r\}$ and $\mathsf{ran}(f)$ for $f(\mathcal{V})$. We take function application to be strict: $f(g(x))\downarrow$ implies $g(x)\downarrow$.

3.1 Definition [Lenses]: A *lens* l comprises a partial function $l\nearrow$ from $\mathcal{V}$ to $\mathcal{V}$, called the *get function* of l, and a partial function $l\searrow$ from $\mathcal{V} \times \mathcal{V}$ to $\mathcal{V}$, called the *putback function*.

The intuition behind the notations $l\nearrow$ and $l\searrow$ is that the *get* part of a lens "lifts" an abstract view out of a concrete one, while the *putback* part "pushes down" a new abstract view into an existing concrete view. We often say "put a into c (using l)" instead of "apply the *putback* function (of l) to (a, c)."

3.2 Definition [Well-behaved lenses]: Let l be a lens and let C and A be subsets of $\mathcal{V}$. We say that l is a *well behaved* lens from C to A, written $l \in C \rightleftharpoons A$, if it maps arguments in C to results in A and vice versa

$$l\nearrow(C) \subseteq A \qquad \text{(GET)}$$
$$l\searrow(A \times C) \subseteq C \qquad \text{(PUT)}$$

and its *get* and *putback* functions obey the following laws:

$$l\searrow(l\nearrow c,\ c) \sqsubseteq c \quad \text{for all } c \in C \qquad \text{(GETPUT)}$$
$$l\nearrow(l\searrow(a,\ c)) \sqsubseteq a \quad \text{for all } (a, c) \in A \times C \qquad \text{(PUTGET)}$$

An example of a lens satisfying PUTGET but not GETPUT is the following. Suppose $C = \texttt{string} \times \texttt{int}$ and $A = \texttt{string}$, and define l by:

$$l\nearrow(s,n) = s \qquad\qquad l\searrow(s',\,(s,n)) = (s',0)$$

Then $l\searrow(l\nearrow(s,1),\,(s,1)) = (s,0) \not\sqsubseteq (s,1)$. Intuitively, the law fails because the *putback* function has "side effects": it modifies information in the concrete view that is not reflected in the abstract view.

An example of a lens satisfying GETPUT but not PUTGET is the following. Let $C = \texttt{string}$ and $A = \texttt{string} \times \texttt{int}$, and define l by :

$$l\nearrow s = (s,0) \qquad\qquad l\searrow((s',n),\,s) = s'$$

PUTGET fails here because some information contained in the abstract view does not get propagated to the new concrete view. For example, $l\nearrow(l\searrow((s',1),\,s)) = l\nearrow s' = (s',0) \not\sqsubseteq (s',1)$.

For now, a simple example of a lens that is well behaved but not very well behaved is as follows. Consider the following lens, where $C = \texttt{string} \times \texttt{int}$ and $A = \texttt{string}$. The second component of each concrete view intuitively represents a version number.

$$l\nearrow(s,n) = s \qquad\qquad l\searrow(s,\,(s',n)) = \begin{cases}(s,n) & \text{if } s = s'\\ (s,n{+}1) & \text{if } s \neq s'\end{cases}$$

The *get* function of l projects away the version number and yields just the "data part." The *putback* function overwrites the data part, checks whether the new data part is the same as the old one, and, if not, increments the version number. This lens satisfies both GetPut and PutGet but not PutPut, as we have $l\searrow(s,\,l\searrow(s',\,(c,n))) = (s,n+2) \not\sqsubseteq (s,n+1) = l\searrow(s,\,(c,n))$.