Lecture 110 of Advanced Programming

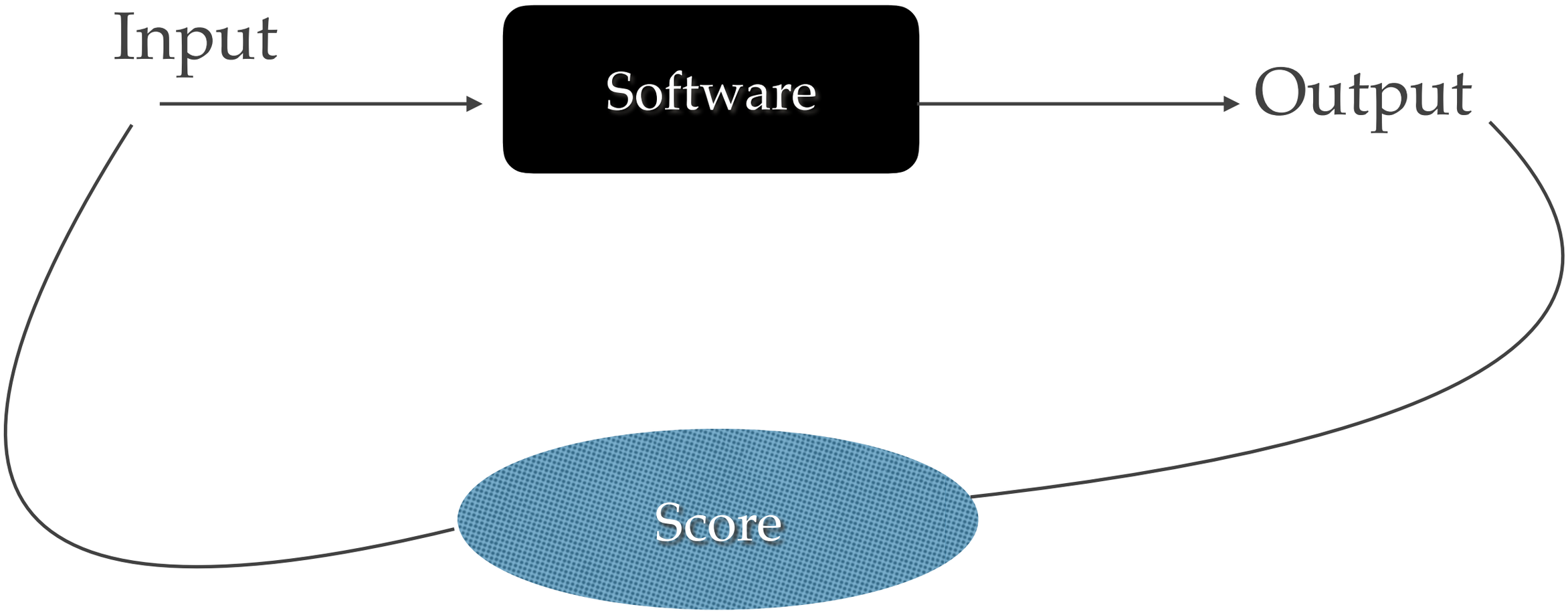
November 14, 2019

Lenses

Andrzej Wasowski & Zhoulai Fu

IT University of Copenhagen

Automated Testing



```
Inputs = []

def score (x):
   if x is already in Inputs:
      return 0;  //lowest score
   else:
      ...
      update Inputs to Inputs + {x}
```

```
Inputs = []

def score (x):
    if x is already in Inputs:
       return 0;  //lowest score
    else:
       ...
       update Inputs to Inputs + {x}
```

Reasoning about "score" was painful — score(x) can have different value for the same x !

Virtues of being immutable

- Safer thread
- Understandable code
- Easier to test and reason

Immutability with case classes in Scala

```
case class Street(name: String, number: Int)
case class Address(country: String, city: String, street: Street)
```

Demo

Scala Quiz: When you create a *case class*, a _____ method is generated for your case class allowing you to return objects with modified fields.

Scala Quiz: When you create a *case class*, a `copy` method is generated for your case class allowing you to return objects with modified fields.

```
case class Street(name: String, number: Int)
case class Address(country: String, city: String, street: Street)
```

Demo

Try this exercise in a group of 2-3 (5 minutes)

```
case class Street(name: String, number: Int)
case class Address(country: String, city: String, street: Street)
```

Exercise

- Create a street
- Create an address based on the street
- Change the street number

Demo

What if the field is further 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 reaction

- 1. FP programming sucks
- **2. We should 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?
  - v.copy(i = 6)
  - Lens automates, and generalizes ”copy” to deal with nested fileds

Monocle Lens

- Monocle is a scala library
- - import monocle.Lens
- - To change the “a” filed of the “case class A(a:Int)”:
  - val l=Lense[A,Int](getter)(setter)
  - - getter: A=>Int
  - - setter: Int =>A =>A

Demo

Laws are useful for testing and reasoning

"Property-based law testing is one of the most powerful tools in the scala ecosystem. "

- x * x >= 0 holds for reals
- (a + b) + c = a + (b + c) holds for integers
- read(write(a,i,v),i)=v holds for arrays

Law of get-set for Lens

if you `get` a value from a and `set` it back in, the result is an object identical to the original a.

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

Demo

Law of set-get for Lens

if you `set` a value, you always `get` the same value back

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

Demo

Laws for Lens

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.

From Monocle package homepage

How laws help computer scientists reasoning

Are there array a, integers b and c that satisfy the following constraint?

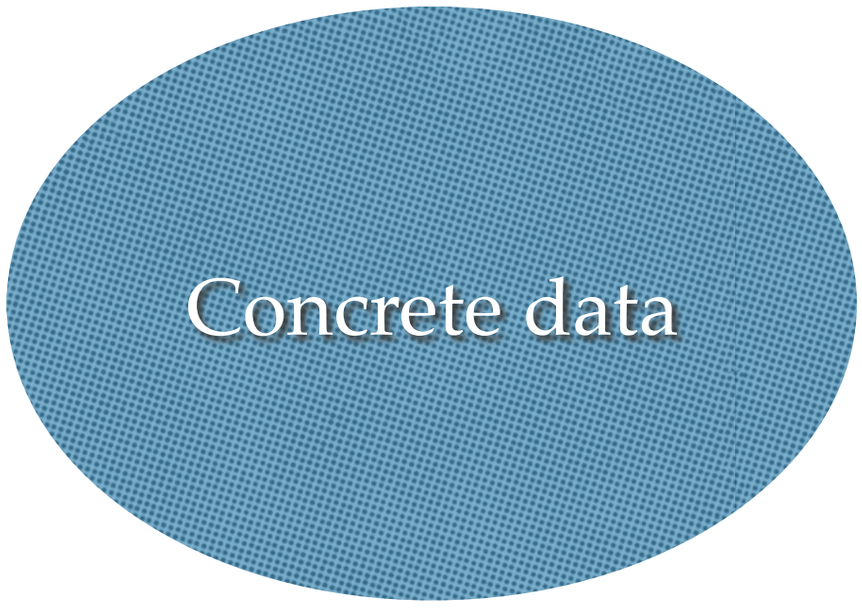
$$b + 2 = c, \quad f(read(write(a,b,3), c\text{-}2)) \neq f(c\text{-}b+1)$$

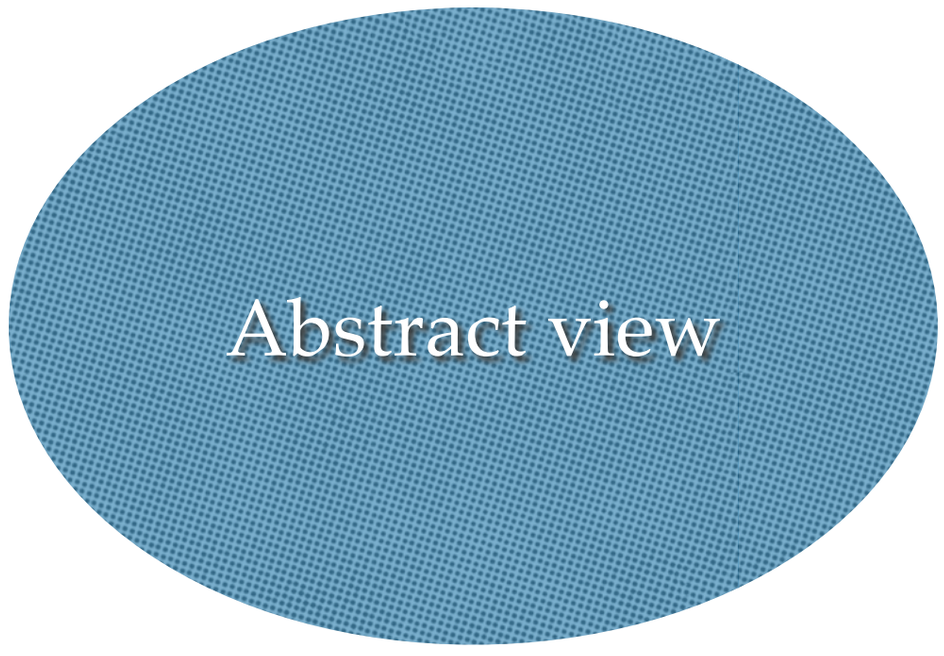
Demo on Blackboard

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





The view-update problem:

Update on View must be correctly reflected on updates on Data

Concrete data

Abstract view

$$c = \left\{ \begin{array}{l} \texttt{Pat} \mapsto \left\{ \begin{array}{l} \texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{http://pat.com} \end{array} \right\} \\ \texttt{Chris} \mapsto \left\{ \begin{array}{l} \texttt{Phone} \mapsto \texttt{888-9999} \\ \texttt{URL} \mapsto \texttt{http://chris.org} \end{array} \right\} \end{array} \right\}$$

$$a = \left\{ \begin{array}{l} \texttt{Pat} \mapsto \texttt{333-4444} \\ \texttt{Chris} \mapsto \texttt{888-9999} \end{array} \right\}$$

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)."

Demo on Blackboard

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:

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

Demo on Blackboard

Conclusions

- We want “copy” for the sake of immutability
- We want “lens” to avoid nested copy
- Monocle is a Scala implementation
- Laws are to be obeyed for using lens
- The bigger picture is the view-update problem [foster07, morris12]

Backup slides

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, $l\nearrow(l\searrow((s',1),\,s)) = l\nearrow s' = (s',0) \not\sqsubseteq (s',1)$.