Lenses

Today's lecture introduces **lenses**, and is inspired by the **lens-tutorial**.

The lens abstraction, and related abstractions, make the concept of a field of an abstraction, a first class notion. It is a little language of its own, uses nice type trickery, and certainly has a learning curve. But when well-understood, it allows for concice, expressive code, and opens new ways of abstraction. It is worth considering in every medium-to-large sized project that handles deep structured data. It is also worth learning because some interesting libraries, such as `diagrams`, make heavy use of it.

Getters and Setters

Here is a product type with a bunch of fields:

```
data Atom = Atom { _element :: String, _point :: Point }
data Point = Point { _x :: Double, _y :: Double }
```

Getters

Haskell's record syntax makes it rather easy to reach deeply inside such a data structure. For example, if we want to get the `x`-position of an atom, we can write

```
getAtomX :: Atom -> Double
getAtomX = _x . _point
```

So the record accessors serve as *getters*, and if we want to reach deeply into a data structure, we can compose these getters. Of course, this is just syntactic sugar, and if we would not have used record syntax, we could easily implement `_x` and `_point` by hand.

Setters

Setting a value is not so easy. There is the record-update syntax that allows us to write the following (but again, the record-update is just syntactic sugar, and we could have written the same with regular pattern matching):

```
setPoint :: Point -> Atom -> Atom
setPoint p a = a { _point = p }
setElement :: String -> Atom -> Atom
setElement e a = a { _element = e }
setX, setY:: Double -> Point -> Point
setX x p = p { _x = x }
setY y p = p { _y = y }
```

Unfortunately, these setters do not compose well, as we see when we want to write a function that sets the `x` of an atom:

```
setAtomX :: Double -> Atom -> Atom
setAtomX x a = setPoint (setX x (_point a)) a
```

In order to compose `setAtomPoint` with `setPointX`, we also need a getter to get the point of the atom!

So it seems that getters and setters are closely related, and we want to bundle them and work with them together.

A simple lens

So let us create an abstract data type that combines the getter and setter of a field, and let us call that a *lens*, as it “zooms into” a field:

```
data Lens a b = { view :: a -> b
                , set :: b -> a -> a
                }
```

The implementations are straight forward:

```
point :: Lens Atom Point
point = Lens _point setPoint
element :: Lens Atom String
element = Lens _element setElement
x, y :: Lens Point Double
x = Lens _x setX
y = Lens _y setY
```

In order to implement `setAtomX`, we want to compose two lenses, and we can do that using a general operator:

```
comp :: Lens a b -> Lens b c -> Lens a c
comp l1 l2 = Lens (view l2 . view l1)
                  (\c a -> set l1 (set l2 c (view l1 a)) a)

setAtomX :: Double -> Atom -> Atom
setAtomX = set (point `comp` x)
```

Modify

In the code that we have just written, there is a very common pattern: Applying a function to a field. And clearly, we can implement that using a getter and a setter:

```
over :: Lens a b -> (b -> b) -> (a -> a)
over l f a = set l (f (view l a)) a
```

So if we want to move an atom to the right, we can simply write

```
moveAtom :: Atom -> Atom
moveAtom = over (point `comp` x) (+1)
```

We can also rewrite `comp`:

```
comp :: Lens a b -> Lens b c -> Lens a c
comp l1 l2 = Lens (view l2 . view l1)
                  (\c -> over l1 (set l2 c))
```

Efficiency

Unfortunately, this is not very efficient. Function `over` uses the lens `l` twice: Once to get the value, and once again to set it. And since `over` is used in `comp`, if we nest our lenses a few layers deep, this gets inefficient very quickly.

How can we fix this? We make `over` primitive!

```
data Lens a a = { view :: a -> b
                , set :: b -> a -> a
                , over :: (b -> b) -> (a -> a)
                }
```

In order to update our existing primitive lenses, we implement a small helper function to derive the `over` code, instead of writing it by hand every time.

```
mkLens :: (a -> b) -> (b -> a -> a) -> Lens a b
mkLens view set = Lens view set over
  where over a = set (f (view a)) a

point :: Lens Atom Point
point = mkLens _point setPoint
element :: Lens Atom Element
element = mkLens _element setElement
x, y :: Lens Atom Double
x = mkLens _x setX
y = mkLens _y setY
```

Now the composition operator uses every lens only once. Good!

```
comp :: Lens a b -> Lens b c -> Lens a c
comp l1 l2 = Lens (view l2 . view l1)
                  (\c -> over l1 (set l2 c))
                  (over l1 . over l2)
```

In fact, with `over` as the primitive notion, there is not need for `set` any more, as that can be implemented with `over`:

```
set :: Lens a b -> b -> a -> a
set l x = over l (const x)
```

Towards van Laarhoven lenses

This is nice, but what if we want to do an *effectful* update? For example, this code does not typecheck:

```
askX :: Atom -> IO Atom
askX a = over (point `comp` x) askUser a
  where
    askUser :: Double -> IO Double
    askUser = do
        putStrLn $ "Current position is " ++ show x ++ ". New Position?"
        answer <- getLine
        return (read answer)
```

`IO` as motivation

Of course we could rewrite it again to use `view` before any `IO` actions, and `set` afterwards, but then we would again be traversing the data structure towards the position of interest twice.

We can fix this as we did before, by allowing a variant of `over` that does `IO`:

```
data Lens a b = { view :: a -> b
                , over :: (b -> b) -> (a -> a)
                , overIO :: (b -> IO b) -> (a -> IO a)
                }

mkLens :: (a -> b) -> (b -> a -> a) -> Lens a b
mkLens view set = Lens view over overIO
  where over a = set (f (view a)) a
        overIO a = do
          b' <- f (view a)
          return $ set b' a


comp :: Lens a b -> Lens b c -> Lens a c
comp l1 l2 = Lens (view l2 . view l1)
```

```
                    (\c -> over l1 (set l2 c))
                    (over l1 . over l2)
                    (overIO l1 . overIO l2)
```

Fancy how composition is so simple again!

Enter the `Functor`

But clearly, we want to do this trick not just for `IO`, but for many type constructors. Some of which might not be `Monads`. So if we look closely at the code for `overIO`, we see that all we really need is a functor instance. So let us generalize this:

```
data Lens a b = { view :: a -> b
                , over :: (b -> b) -> (a -> a)
                , overF :: forall t. Functor t => (b -> t b) -> (a -> t a)
                }

mkLens :: (a -> b) -> (b -> a -> a) -> Lens a b
mkLens view set = Lens view over overF
  where over a = set (f (view a)) a
        overF a = (\b' -> set b' a) <$> f (view a)
```

The `forall t` says that the function stored in the `overF` field of a `Lens` works with any functor `t` that you want to use it at.

Getting rid of `over`

But look at how similar the type signatures of `overF` and `over` are. If we could somehow make the `t` go away, they would be identical, and `overF` would be enough?

So we want a type constructor `t` that is equal to its argument.

```
type I x = x
instance Functor I where
  fmap f x = f x
```

is a good start, but we cannot define a `Functor` instance for that, so we have to use a `newtype` to get the *identitiy functor*:

```
newtype I x = MkI x

unI :: I x -> x
unI (MkI x) = x

instance Functor F where
  fmap f x = MkI (f (unI x))
```

With this particular `Functor` instance, we can derive `over` from `overF` and remove it from the `Lens` type:

```
over :: Lens a b -> (b -> b) -> (a -> a)
over l f a = unI $ overF l f' a
  where f' b = MkI (f b)
```

Getting rid of `view`

That was nice, we are again down to two primitive operations. Can we do better? Is, maybe, in some way, `view` also an instance of `overF`?

If we try to make the type match, from right to left, it might work if `t a` would somehow be `b` – then at least we would have `a -> b` a the end, as desired.

We would somehow have to provide a function `b -> t b` though, that works in every case. Since we get to pick `t`, why not make it always `b`:

```
newtype C b x = MkC b

unC :: C b x -> b
unC (MkC b) = b

instance Functor C where
  fmap f (MkC b) = MkC b
```

With this *constant functor*, we can define `view` in terms of `overF`:

```
view :: Lens a b -> a -> b
view l a = unC $ overF l MkC a
```

`Lens` is just a type synonym

But now `Lens` has become a product type with only one field. This menas that the type `Lens a b` is *isomporphic* to the type `forall t. Functor t => (b -> t b) -> (a -> t a)`. In that case, why bother with the `Lens` constructor and the `overF` field name at all? We can get rid of them!

```
type Lens a b = forall t . Functor t => (b -> t b) -> (a -> t a)
```

Interestingly, now

```
comp :: Lens a b -> Lens b c -> Lens a c
comp l1 l2 = l1 . l2
```

so we can get rid of this function as well, and use plain old function composition `.`!

Traversals

Where there is a `Functor`, an `Applicative` cannot be far. What if we do change the constraint in the lens type:

```
type Traversal a b = forall t . Applicative t => (b -> t b) -> (a -> t a)
```

The name `Traversal` will become clear later. The first thing we notice is that every `lens` is a traversal, becauase every `Applicative` is a `Functor`:

```
lensToTraversal :: Lens a b -> Traversal a b
lensToTraversal l = l
```

I wrote this function only to show you that the types check, but we can just use a lens as a traversal directly!

The other direction does not work, because not every `Functor` is an `Applicative`.

Generalizing `over`

So whatever a `Traversal` is, it is more general than a `Lens`. Thus, if we can change some of our functions to take a `Traversal` instead of a `Functor`, then the world is strictly a better place.

Can we change the type of `over` as follows?

```
over :: Traversal a b -> (b -> b) -> (a -> a)
over l f a = unI $ l f' a
  where f' b = MkI f b
```

Yes we can! Well, almost, the compiler wants us to provide an `Applicative` instance for `I`. Fine with me:

```
instance Applicative I where
  pure x = MkI x
  f <$> x = MkI $ (unI f) (unI x)
```

Since `set` is just defined in terms over `over`, we can now also relax the type signature of `set` to use `Traversal`.

Non-Lens traversals

Can we do the same thing with `view`? No, we cannot! The constant functor is not applicative (there is no way of implementing `pure :: a -> C b a`).

So a `Traversal a b` describes how one can (possibly effectful) set or update values of type `b` in `a` (like `Lens`), but not get a value of type `b` (unlike `Lens`). If we try to think of concrete `a` where that is the case, what come to mind?

For example `Maybe b`! We certainly can apply a function to the contained thing, if it is there, and maybe even with effect:

```
this :: Traversal (Maybe a) a
this f Nothing  = pure Nothing
this f (Just x) = Just <$> f x
```

Here, we cannot expect to have a `view` because not every `Maybe a` has an `a`.

Another example would be lists:

```
elems :: Traversal [a] a
elems f []     = pure Nothing
elems f (x:xs) = (:) <$> f x <*> elems f xs
```

Here we cannot expect to have a useful `view` because a `[a]` might not have an `a`, but also because it might have many.

Getting all of them

So we cannot have `view` because the structure might have zero or more than one elements. Well, then at least we should be able to get a list of them?

```
listOf :: Traversal a b -> a -> [a]
```

Again, we can try to implement that using a suitable `Functor`. We compare the desired type with the type of a `Traversal` and find that we again need a constant functor, this time, though, storing a list of `b`s:

```
newtype CL b x = MkC [b]

unCL :: CL b x -> [b]
unCL (MkCL b) = b

instance Functor CL where
  fmap f (MkCL b) = MkCL (map f b)

instance Applicative CL where
  pure _ = MkCL []
  MkCL bs1 <*> MkCL bs2 = MkCL (bs1 ++ bs2)
```

With this *constant functor*, we can define `view` in terms of `overF`:

```
listOf l a = unCL $ overF l MkCL a
```

(In reality one would use `Const [b] x` here with the `Applicative` instance for `Const` with the `Monoid` constraint on the first argument of `Const`, but since we did not discuss `Monoid` in this class, we do it by hand here.)

What is a `Traversal` now?

Similar to `Lens` is one position in a data structure (and precisely one, and one that is always there), `Traversal` describes many position in a data structure.

And since `Lens` and `Traversal` compose so nicely, you can describe pretty complex "pointer" well. For example with **xml-lens**, this `Traversal` extracts the title of all books with a specific category from an XML fragment.

```
root . el "books" ./ el "book" . attributeIs "category" "Textbooks" ./ el "title" . text
```

Further reading

The story presented here is rather simple. If you look at the **lens libray** you see more abstractions (`Prism`, `Iso`, etc.). This library also comes with a large number of concrete lenses, traversals etc for many data structures, and has cool tricks so that `_2` for example is a lens for the second element of a tuple, for any tupel size.

In that package, what we called `Lens` and `Traversal` is actually called `Lens'` and `Traversal'`, and the version without quote allows `over` to change the type of the thing pointed at.

But note that even in the `lens` library, all these notions are just type synonyms, so you can define lenses as we did, without using a library, and you are still compatible with these libraries! Also see **`microlens`** for a library, compatible with `lens`, but smaller, less dependencies and better documented.