The Overview of Lens

its Implmentation and Application

2017-10-20

Haoran Sun
ZJU Lambda
shuenhoy@zju.edu.cn

Part I: Overview

Motivation

You are developing a RPG and you get this:

```
data Hero = Hero {
  heroLevel :: Int, weapon :: Weapon
}

data Weapon = Weapon {
  basicAttack :: Int, weaponLevel :: Int, magicGem :: Gem
}
data Gem = Gem {
  gemLevel :: Int,
  gemName :: String
}
```

Also we have

```
setHeroLevel :: Hero -> Int -> Hero
setWeapon    :: Weapon -> Hero -> Hero
...
```

Motivation

Getting the inner value

```
gemLevel.magicGem.weapon $ hero
-- Or
hero & (weapon>>>magicGem>>>gemLevel)
```

Ok… not so annoying?

Motivation

What about setting the inner value?

```
hero' = hero {
    weapon = (weapon hero) {
        magicGem = (magicGem.weapon $ hero){
            gemName = "WTF" }}}
```

WTF.

How to use Lens?

Let's see the lens version.

```
view (weaponLens.magicGemLens.gemLevelLens) hero

hero'  = set (weaponLens.magicGemLens.gemNameLens) "Gem" hero

hero'' = over (weaponLens.magicGemLens.gemLevelLens) (+1) hero
```

Much better.

How to use Lens?

```
hero .^ weaponLens.magicGemLens.gemLevelLens

hero'  = hero & weaponLens.magicGemLens.gemNameLens .~ "Gem"

hero'' = hero & weaponLens.magicGemLens.gemLevelLens) %~ (+1)
```

Even Better.

Now we can play with complex datastructure happliy ☺

What's going on?

See the code again.

```
hero .^ weaponLens.magicGemLens.gemLevelLens

hero'  = hero & weaponLens.magicGemLens.gemNameLens .~ "Gem"
```

And this.

```
hero.weapon.magicGem.gemLevel
hero.weapon.magicGem.gemName = "Stone"
```

Similar looks?

What's going on?

Don't forget *(.)* here is not anything special.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g x = \x -> g (f x)
```

That means all the *xxLens* are just functions!

And functions can be composed!

Part II: Make a substitute

Try to do something similar (and simpler).

We already have getter and setter for each field.

Now we combine them into a tuple and we can define the compose function.

```
type L a b = (a -> b, b -> a -> a)

(.>) :: L a b -> L b c -> L a c
(g1, s1) .> (g2, s2) = (g2 . g1, \c a -> s1 (s2 c (g1 a)) a)

viewL :: L a b -> a -> b
viewL (g, _) = g

setL :: L a b -> b -> a -> a
setL (_, s) = s

overL :: L a b -> (b -> b) -> a -> a
overL (g, s) f a = s (f $ g a) a
```

Try to do something similar (and simpler).

Now we can easily get *xxxL* like this:

```
weaponL   = (weapon, setWeapon)
gemLevelL = (gemLevel, setGemLevel)
```

Try to use it.

```
viewL (weaponL.>magicGemL.>gemLevelL) hero
hero' = setL (weaponL.>magicGemL.>gemLevelL) 2 hero
```

Yes! We get a simplifed Lens.

Comaring the Luxury Lens and ours

	Luxury Lens	**Our Lens**
Type	A Unkown Function Type	`(a -> b, b -> a -> a)`
Compose	Function Compose `(.)`	`(.>)`
Use	Almost the same	
Feature	abstract	direct

Part III: Preparations for Lens

- It's a function
- It is related to the object type *a* and field type *b*
- It can be composed with each other
- It contains all the information we need to get/set a data field

Find the Lens type

- It's a function and related to the object type *a* and field type *b*

```
type Lens b a = (???) -> (???)
```

- It can be composed with each other

```
aL :: Lens b a
bL :: Lens a c
aL.bL :: Lens b c
```

So we know the first `(???)` is about type *a* and the second, type *b*

Find the lens type

When we use a Lens, we will need to pass it a type *b* value.

```
type Lens b a = (???) -> (b -> ???)
```

Again, to make it composable.

```
type Lens b a = (a -> ???) -> (b -> ???)
```

And we don't know what to do next:(

Build the *view* first

Recall the usage first.

```
view (weaponLens.magicGemLens.gemLevelLens) hero
```

We can do this.

```
view :: Lens b a -> b -> a
view lens b = lens ??? b
```

So we need

```
type Lens b a = (a -> ???) -> (b -> ???)
```

Build the *view* first

Consider our example.

```
weaponVLens   :: (Weapon -> ???) -> (Hero -> ???)
magicGemVLens :: (Gem -> ???)    -> (Weapon -> ???)
gemLevelVLens  :: (Int -> ???)    -> (Gem -> ???)
```

For *view* , if we want

```
weaponVLens.magicGemVLens.gemLevelVLens :: (Int -> Int) -> (Hero -> Int)
```

all the (???) should be *int* to compose these functions

Build the *view* first

Consider our example.

```
weaponVLens   :: (Weapon -> ???) -> (Hero -> ???)
magicGemVLens :: (Gem -> ???)    -> (Weapon -> ???)
gemNameVLens  :: (Int -> ???)    -> (Gem -> ???)
```

For *view*, if we want

```
weaponVLens.magicGemVLens :: (Gem -> Gem) -> (Hero -> Gem)
```

all the (???) should be *Gem* to compose these functions

Build the *view* first

now we can get the type

```
type VLens b a = forall c. (a -> c) -> (b -> c)
```

The *view* definition

```
viewV vlens b = vlens id b
```

And the VLens

```
wweaponVLens  :: VLens Hero Weapon
weaponVLens f h = f (weapon h)

magicGemVLens :: VLens Weapon Gem
magicGemVLens f h = f (magicGem h)

gemLevelVLens :: VLens Gem Int
gemLevelVLens f h = f (gemLevel h)
```

Build the *over*

Reconsider our example.

```
weaponOLens   :: (Weapon -> ???) -> (Hero -> ???)
magicGemOLens :: (Gem -> ???)    -> (Weapon -> ???)
gemNameOLens  :: (Int -> ???)    -> (Gem -> ???)
```

For *over*, if we want

```
weaponOLens.magicGemOLens :: (Gem -> Gem) -> (Hero -> Hero)
```

all the (???) should be the same as the types in front of them.

Build the *over*

now we can get the type

```
type OLens b a =  (a -> a) -> (b -> b)
```

The *over* definition

```
overO :: OLens b a -> (a -> a) -> b -> b
overO vlens= vlens
```

And the OLens

```
weaponOLens ::  OLens Hero Weapon
weaponOLens f h = (`setWeapon` h) $ f (weapon h)

magicGemOLens :: OLens Weapon Gem
magicGemOLens f w = (`setMagicGem` w) $ f (magicGem w)
```

```
gemLevelOLens :: OLens Gem Int
gemLevelOLens f g = (`setGemLevel` g) $ f (gemLevel g)
```

Build the *set*

Pretty easy!

```
setO :: OLens b a -> a -> b -> b
setO vlens s = vlens (const s)
```

Part IV : Abstract *Lens* from *VLens* and *OLens*

Compare *VLens* and *OLens*

	VLens	**OLens**
Type signature	`forall c. (a -> c) -> (b -> c)`	`(a -> a) -> (b -> b)`
Common Pattern	`(a -> ???) -> (b -> ???)`	
Lens definition	`\f v->f (getter v)`	`\f v->(setter $ f (getter v)) v`
Common Part	`f (getter v)`	
Operation	`viewV vlens = vlens id`	`overO vlens = vlens`

Can we do this?

```
type FLens b a = forall c d. (a -> c) -> (b -> d)
weaponFLens :: FLens Hero Weapon
weaponFLens f h = ???
```

What we need

We want the Lens to behave differently due to the caller function.

Maybe we need a typeclass, it contains some value and we can perform some operations on the value due to specific types.

Sounds familiar?

```
type Lens b a = Functor f => (a -> f a) -> (b -> f b)
```

Rewrite the *over*

We already have this

```
overO :: OLens b a -> (a -> a) -> b -> b
overO vlens= vlens
```

Just wrap the previous *over* with a *Identity* functor, which does nothing extra.

```
over lens f = runIdentity . lens (Identity . f)
```

Rewrite the Lens

We have defined the Lens as follows

```
weaponOLens ::  (a -> a) -> (b -> b)
weaponOLens f h = (`setWeapon` h) $ f (weapon h)
```

Just lift it!

```
weaponLens ::  Functor f => (a -> f a) -> (b -> f b)
weaponLens f h = (`setWeapon` h) <$> f (weapon h)
```

By using *Identity* Functor, the Lens and *over* function behave exactly the same as before.

Traits of *view*

```
type VLens b a = forall c. (a -> c) -> (b -> c)
viewV vlens b = vlens id b
```

- Modify nothing
- Each layer returns what its inner layer return

Const Functor

```haskell
newtype Const a b = Const {getConst :: a}
instance Functor (Const a) where
  fmap f c = c
```

- **Modify nothing**
  fmap just return what it gets and ignores the *f*
- **Each layer returns what its inner layer return**
  getConst always returns the initial value after many `fmap` operation

Construct *view* with *Const* Functor

After all the analysis, we can easily get the *view* definition.

```
view :: Lens b a -> b -> a
view lens b = getConst $ lens Const b
```

Now we can play with data as we have shown!

```
view (weaponLens.magicGemLens.gemLevelLens) hero

hero'  = set (weaponLens.magicGemLens.gemNameLens) "Gem" hero

hero'' = over (weaponLens.magicGemLens.gemLevelLens) (+1) hero
```

Questions

Consider these datastructures:

```
data Hero = Hero {
  heroLevel :: Int, weapon :: Maybe Weapon
}

data Weapon = Weapon {
  basicAttack :: Int, weaponLevel :: Int, magicGem :: Maybe Gem
}
data Gem = Gem {
  gemLevel :: Int,
  gemName :: String
}
```

How to deal with them gently?

Thank You!