Lens

박주형 in 하스켈 학교

목차

Lens없이 get

```haskell
fst (1,2) -- 1
snd (1,2) -- 2
(fst . snd) ((1, 2), ("a", "b"))
```

Lens없이 get

```haskell
fst (1,2) -- 1
snd (1,2) -- 2
(fst . snd) ((1, 2), ("a", "b"))  -- "a"

(fst . snd) :: (a, (c, b)) -> c
```

Lens없이 set

```haskell
fstMap f (x,y) = (f x, y)
sndMap f (x, y) = (x, f y)
fstMap (+1) (1,2) -- (2,2)
(fstMap . sndMap) (+1) ((1,2), (10, 20))
```

Lens없이 set

```haskell
fstMap f (x,y) = (f x, y)
sndMap f (x, y) = (x, f y)
fstMap (+1) (1,2) -- (2,2)
(fstMap . sndMap) (*0) ((1,2), (10, 20))
-- ((1,0), (10, 20))
(fstMap . sndMap) ::
  (b -> d) -> ((a, b), c) -> ((a, d), c)
```

Get과 Set의 compose 순서가 다름

Lens없이 set 2

```haskell
data Game = Game { _player :: Unit } deriving (Show)
data Unit = Unit { _x :: Int } deriving (Show)

playerMap f game@(Game { _player = player }) =
  game { _player = f player}
xMap f unit@(Unit { _x = x }) = unit { _x = f x }

(playerMap . xMap) (+1) (Game (Unit 3))
```

Record의 set 문법이 불편

Lens Get / Set

Tuple

```haskell
view _1 (1,2) -- 1
set _1 "hs" ((),1) -- ("hs", 1)
over _1 (+1) (1, 2) -- (2,2)
```

Lens Get / Set

compose

```
cplx = (((1,2), "st"), (3,4))
view (_1._1._2) cplx -- 2
set (_1._1._2) "hs" cplx -- (((1,"hs"), "st"), (3,4))
over (_1._1._2) (*3) cplx -- (((1,6),"st"),(3,4))
```

Lens Get / Set

Record

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens
data Game = Game { _player :: Unit } deriving (Show)
data Unit = Unit { _x :: Int } deriving (Show)
makeLenses ''Game
makeLenses ''Player
initialGame = Game $ Unit $ 3
view player initialGame -- Unit {_x = 3}
view (player.x) initialGame -- 3
over (player.x) (+3) initialGame
 -- Game {_player = Unit {_x = 6}}
```

렌즈의 장점

1. 일관성 있는 composing
2. 편리한 record 수정

Lens operators

```haskell
(1,2)^._1 -- 1
((),1) & _1 .~ "hs" -- ("hs", 1)
(1,2) & _1 %~ (+1) -- (2,2)
```

Lens operators

```haskell
cplx = (((1,2), "st"), (3,4))

cplx^._1._1._2 -- 2 (view)
cplx & _1._1._2 .~ "hs" -- (((1,"hs"), "st"), (3,4)) (set)
cplx & _1._1._2 %~ (*3) -- (((1,6),"st"),(3,4)) (over)

cplx & _1._1._2 .~ "hs"
     & _2 .~ "removed"
     & _1._1._1 %~ (*9) -- (((9,"hs"), "st"), "removed")
```

Lens operators

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens
data Game = Game { _player :: Unit } deriving (Show)
data Unit = Unit { _x :: Int } deriving (Show)
makeLenses ''Game
makeLenses ''Player
initialGame = Game $ Unit $ 3
initialGame^.player.x -- 3
initialGame & player.x .~ 19
 -- Game {_player = Unit {_x = 19}}
initialGame & player.x %~ (+3)
 -- Game {_player = Unit {_x = 6}}
```

목차

1. Lens 사용하기

2. 타입을 이해하기

- Identity & Over
- Const & View
- Traversal
- Prism
- State Monad

Type of Lens

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

Identity

```haskell
newtype Identity a = Identity { runIdentity :: a }

fmap :: (a -> a) -> Identity a -> Identity a
fmap f (Identity value) = Identity (f value)
```

Over

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```
over :: Lens' s a -> (a -> a) -> s -> s
over lens modifier container =
  lens (Identity . modifier) container & runIdentity
```

Over

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```
over :: Lens' s a -> (a -> a) -> s -> s
over lens modifier container =
  lens (Identity . modifier) container & runIdentity
```

```
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

Over

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```
over :: Lens' s a -> (a -> a) -> s -> s
over lens modifier container =
  lens (Identity . modifier) container & runIdentity
```

```
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

```
over _1 (+1) (1,2)
  = (_1) (Identity . (+1)) (1,2) & runIdentity
```

Over

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```
over :: Lens' s a -> (a -> a) -> s -> s
over lens modifier container =
  lens (Identity . modifier) container & runIdentity
```

```
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

```
over _1 (+1) (1,2)
  = (_1) (Identity . (+1)) (1,2) & runIdentity
  = (,2) <$> (Identity . (+1)) 1 & runIdentity
```

Over

```haskell
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```haskell
over :: Lens' s a -> (a -> a) -> s -> s
over lens modifier container =
  lens (Identity . modifier) container & runIdentity
```

```haskell
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

```haskell
over _1 (+1) (1,2)
  = (_1) (Identity . (+1)) (1,2) & runIdentity
  = (,2) <$> (Identity . (+1)) 1 & runIdentity
  = (,2) <$> (Identity 2) & runIdentity
```

Over

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```
over :: Lens' s a -> (a -> a) -> s -> s
over lens modifier container =
  lens (Identity . modifier) container & runIdentity
```

```
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

```
over _1 (+1) (1,2)
  = (_1) (Identity . (+1)) (1,2) & runIdentity
  = (,2) <$> (Identity . (+1)) 1 & runIdentity
  = (,2) <$> (Identity 2) & runIdentity
  = Identity (2,2) & runIdentity = (2,2)
```

Const

```haskell
newtype Const a b = Const { getConst :: a }

instance Functor (Const m) where
    fmap _ (Const v) = Const v
```

View

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```
view :: Lens' s a -> s -> a
view lens container =
  lens Const container & runConst
```

View

```haskell
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```haskell
view :: Lens' s a -> s -> a
view lens container =
  lens Const container & runConst
```

```haskell
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

View

```haskell
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```haskell
view :: Lens' s a -> s -> a
view lens container =
  lens Const container & runConst
```

```haskell
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

```haskell
view _1 (1,2)
  = (_1) Const (1,2) & runConst
```

View

```haskell
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```haskell
view :: Lens' s a -> s -> a
view lens container =
  lens Const container & runConst
```

```haskell
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

```haskell
view _1 (1,2)
  = (_1) Const (1,2) & runConst
  = (,2) <$> (Const 1) & runConst
```

View

```haskell
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

```haskell
view :: Lens' s a -> s -> a
view lens container =
  lens Const container & runConst
```

```haskell
_1 :: (a -> f a) -> (a,b) -> f (a,b)
_1 mapper (x,y) = (,y) <$> mapper x
```

```haskell
view _1 (1,2)
  = (_1) Const (1,2) & runConst
  = (,2) <$> (Const 1) & runConst
  = (Const 1) & runConst = 1
```

Traversal

Data가 여러개일 때 처리하기

```haskell
itmes :: Traversal' [a] a
items = tranverse

toListOf items [1,2,3] -- [1,2,3]
over items (+1) [1,2,3] -- [2,3,4]
set items 0 [1,2,3] -- [0,0,0]
```

Traversal

```haskell
data User = User { posts :: [Post] } deriving (Show)
data Post = Post { title :: String } deriving (Show)

posts :: Lens' User [Post]
posts f (User ps) = User <$> f ps
title f (Post _title) = Post <$> f _title

user = (User [Post "a", Post "b"])
toListOf (posts.traverse.title) user -- [ "a", "b" ]
over (posts.traverse.title) (++"c") user
-- User{_posts = [Post{_title = "ac"},Post{_title = "bc"}]

user^..posts.traverse.title -- ["a", "b"]
```

Traversal

traverse?

Prelude 에 있는 Traversable t 의 함수

```
traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
type Lens' s a =
  forall f. Functor f => (a -> f a) -> s -> f s
```

Prism

```haskell
_Left :: Prism' (Either a b) a
_Right :: Prism' (Either a b) b

preview _Left (Left "HI") -- Just "HI"
preview _Left (Right 3) -- Nothing
review _Left "hi" -- Left "HI"
set _Left 1 (Left 3) -- Left 1
set _Left 1 (Right 3) -- Right 3
over _Left (+1) (Left 3) -- Left 4
over _Left (+1) (Right 3) -- Right 3
```

Prism

```haskell
data FooBarBaz a
  = Foo
  | Bar a
  | Baz Int Char
  | Deep ((Int, Int), String) deriving (Show)
makePrisms ''FooBarBaz

preview _Foo (Foo 3) -- Just 3
preview _Foo (Bar "a") -- Nothing
(Deep ((1,2),"as"))^?_Deep._1._2
```

State Monad

```haskell
doInState :: StateT ((Int, Int), Int) IO ()
doInState = do
  assign (_1._2) 3
  _1._1 .= 7

execStateT doInState ((1,2),3) -- ((7,3),3)
```

State Monad

```
execStateT (_1._2 *= 10) ((1,2),3) -- ((1,20),3)
execStateT (_1._2 += 10) ((1,2),3) -- ((1,12),3)
execStateT (_1._2 ^= 10) ((1,2),3) -- ((1,1024),3)
execStateT (_1._2 //= 2) ((1,2),3) -- ((1,1),3)
```

Operators

```haskell
-- (^.) = view
((1,2), 3)^._1._2 -- 2
-- (.~) = set
((1,2), 3) & _1._2 .~ 10 -- ((1,10), 3)
-- (%~) = over
((1,2), 3) & _1._2 %~ (*10) -- ((1, 20), 3)

-- (^..) = toListOf
[(1,2), (3,4)]^..traverse._1 -- [1,3]
-- (^?) = preview
(Left (1,3))^?_Left._2 -- Just 3
-- (.=) = Set in StateT
execStateT (_1._2 .= 10) ((1,2),3) -- ((1,10),3)
```

Operators

```haskell
-- (+~) = += operator
((1,2),3) & _1._2 +~ 3 -- ((1,5),3)
-- (.+) = += operator in monad
execStateT (_1._2 += 10) ((1,2),3) -- ((1,12),3)

-- (^~), (||~), (&&~), (*~), (<>~)
-- (^=), (||=), (&&=), (*=), (<>=)
```

Reference

- http://www.scs.stanford.edu/16wi-cs240h/slides/lenses-slides.html
- http://kseo.github.io/posts/2016-12-10-encodings-of-lense.html
- http://blog.jakubarnold.cz/2014/07/14/lens-tutorial-introduction-part-1.html
- http://blog.jakubarnold.cz/2014/08/06/lens-tutorial-stab-traversal-part-2.html

Reference

- https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial
- http://www.haskellforall.com/2013/05/program-imperatively-using-haskell.html