Линзы в Haskell

Романовский А. А.

Содержание

- Мотивация
- Концепция линзы
- Тип линзы
- Новый тип линзы!
- Операторы для работы с Lens
- Композиция линз
- Снова новый тип линзы!
- Полезные ресурсы

Мотивация

```
public class Person
{
    private String name;
    private int age;
```

```
public String Name
{
    get{ return name;}

    set
    {
        if (value == "") throw new ArgumentException("Error! Empty
          name!");
        name = value;
    }
}

public int Age
{
    get { return age; }
    set
    {
        if (value < 0) throw new ArgumentException("Error! Negative
          age!");
        age = value;
    }
}
```

```
var ivan = new Person("Ivan", 15);
ivan.Name = "42";
ivan.Age += 1;
Console.WriteLine(ivan.Name); // 42
Console.WriteLine(ivan.Age); // 16
ivan.Age = -100; //EXCEPTION!!
```

Концепция линзы

Person

Концепция линзы (в типах)

s

a

Тип линзы

Линза = Геттер + Сеттер

```
view :: s -> a

set :: a -> s -> s
```

```
viewName :: Person -> String

setName :: String -> Person -> Person
```

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

Пример

```haskell
data Person = Person {name :: String, age :: Int} deriving (Show)

setName :: String -> Person -> Person
setName n Person{name = nm, age = a} = Person {name = n, age = a}

nameL :: Lens' Person String
nameL = Lens' {view = name, set = setName}

p = Person {name = "Ivan", age = 15}

pn = view nameL p

--"Ivan"

p2 = set nameL "Vasya" p

--Person {name = "Vasya", age = 15}
```

Более сложное преобразование

```
(a -> a) -> s -> s
```

set :: a -> s -> s

```
over' :: Lens' s a -> (a -> a) -> s -> s
over' l f s = set l (f x) s
  where x = view l s
```

Приходится использовать и view и set!

```
p3 = over' nameL (++" Ivanovich") p
--Person {name = "Ivan Ivanovich", age = 15}
```

А если бы вычисление было с контекстом?

```
over'' :: Lens' s a -> (a -> Maybe a) -> s -> Maybe s

over''' :: Lens' s a -> (a -> Either String a) -> s -> Either String s

over'''' :: Lens' s a -> (a -> (Int, a)) -> s -> (Int, s)
```

Новый тип линзы!

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

1) Что, если контекста нет? ((a -> a) -> s -> s)

2) Где сеттер?

3) Где геттер?

Особенность в том, что особенностей нет

```
newtype Identity a = Identity a
```

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

```
instance Functor Identity where
    f `fmap` (Identity a) = Identity $ f a
```

```haskell
over :: Lens' s a -> (a -> a) -> s -> s
over l f s = runIdentity $ l id_f s -- l id_f s :: Identity s
  where --id_f :: a -> Identity a
    id_f x = Identity $ f x
```

```haskell
--over :: Lens' s a -> ((a -> a) -> s -> s)
over :: Lens' s a -> (a -> a) -> s -> s
over l f = runIdentity . l (Identity . f)
```

Сеттер

```
set :: Lens' s a -> a -> s -> s
set ln x s = runIdentity $ ln set_fld s -- ln set_fld s :: Identity s
  where
    set_fld _ = Identity x --set_fld :: a -> Identity a
```

Лирическое отступление (трюки с типами)

```
data Either a b = Left a | Right b
```

Значение, формально имеющее тип Either a b, может на самом деле не содержать в себе значение типа a или значение типа b

```
x = Left 4 :: Either Int String

y = Right "hello" :: Either Int String
```

```
instance Functor (Either a) where
 -- fmap :: (b -> c) -> (Either a) b -> (Either a) c
   fmap f (Left x) = Left x -- f :: b -> c, x :: a
   fmap f (Right x) = Right $ f x
```

Hack the Gibson

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

newtype Const a s = Const a

newtype Const a s = Const {getConst :: a}

instance Functor (Const a) where
   fmap f (Const x) = Const x
```

Как отсюда вытащить тип a ?

f :: s -> t

:: Const a s   :: Const a t

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

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

Const a   Const a

```haskell
view :: Lens' s a -> s -> a
view ln s = getConst $ ln Const s
-- Const :: a -> Const a a
-- Const a :: Const a a
-- ln Const s :: Const a s
```

```haskell
--view :: Lens' s a -> (s -> a)
view :: Lens' s a -> s -> a
view ln = getConst . (ln Const)
```

Пример

```
data Person = Person {_name :: String, _age :: Int} deriving (Show)
```

```
name :: Lens' Person String
name f Person{_name = n, _age = a} = buildP <$> (f n)
  where
    buildP = \nm -> Person {_name = nm, _age = a}
```

```
age :: Lens' Person Int
age f Person{_name = n, _age = a} = buildP <$> (f a)
  where
    buildP = \ag -> Person {_name = n, _age = ag}
```

Компилятор может помочь!

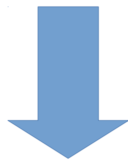
```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens
```

← Из стандартного пакета lens (см. полезные ресурсы)

```haskell
data Person = Person {_name :: String, _age :: Int} deriving (Show)

$(makeLenses ''Person)
```



Только для простых случаев!

```haskell
name :: Lens' Person String

age :: Lens' Person Int
```

Использование

```haskell
p = Person {_name = "Ivan", _age = 15}

p2 = set name "Vasya" p
--Person {_name = "Vasya", _age = 15}

pName = view name p
--"Ivan"

p3 = set age 19 p
--Person {_name = "Ivan", _age = 19}

pAge = view age p
--15

p4 = over age (+1) p
--Person {_name = "Ivan", _age = 16}
```

Вычисления с контекстом

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

Вычисление с контекстом

```
f :: String -> Maybe String
f "Ivan" = Just "Ivan"
f _ = Nothing

p5 = name f p
-- Just (Person {_name = "Ivan", _age = 15})

p6 = name f p2
-- Nothing

f2 :: Int -> [Int]
f2 x = [x - 1, x, x + 1]

p7 = age f2 p
--[Person {_name = "Ivan", _age = 14},
-- Person {_name = "Ivan", _age = 15},
-- Person {_name = "Ivan", _age = 16}]
```

Операторы для работы с Lens

^. ≡ view

```
(^.) :: s -> Lens' s a -> a
s ^. ln = view ln s
```

```
p_age = view age p
```

```
p_age = p ^. age
```

.~ ≡ set

```
(.~) :: Lens' s a -> a -> s -> s
(ln .~ f) s = set ln f s
```

```
p3 = set age 19 p
```

```
p3 = (age .~ 19) p
```

Операторы для работы с Lens

%~ ≡ over

```
(%~) :: Lens' s a -> (a -> a) -> s -> s
(ln %~ f) s = over ln f s
```

```
p4 = over age (+1) p
```

```
p4 = (age %~ (+1)) p
```

Операторы для работы с Lens

Меняем местами функцию и аргумент

```
(&) :: a -> (a -> b) -> b
infixl 1 &
x & f = f x
```

.~ ≡ set

```
p3 = (age .~ 19) p
```

```
p3 = p & age .~ 3
```

%~ ≡ over

```
p4 = (age %~ (+1)) p
```

```
p4 = p & age %~ (+1)
```

Я вижу то, чего не видят обычные люди

```
newtype Temp = Temp {fahrenheit :: Float}


cToF :: Float -> Float
cToF c = (c + 32) * 9 / 5

fToC :: Float -> Float
fToC f = (f - 32) * 5 / 9



centigrade :: Lens' Temp Float
centigrade centi_fn (Temp faren) = temp_c <$> (centi_fn $ fToC faren)
  where
    temp_c = (\centi -> Temp $ cToF centi)
```

Использование

```
t = Temp 15

c = t ^. centigrade -- -9.444445

t2 = t & centigrade .~ 0
-- t2 ^. centigrade == 0.0
-- t2 & fahrenheit == 32.0
```

Композиция линз

```haskell
  data Address = Address {_street :: String, _house :: Int}


  data Person2 = Person2 {_name :: String, _age :: Int, _addr :: Address}

street :: Lens' Address String
street f Address{_street = s, _house = h} = buildA <$> (f s)
  where
    buildA = \st -> Address{_street = st, _house = h}

addr :: Lens' Person2 Address
addr f Person2{_name = n, _age = a, _addr = ad} = buildP <$> (f ad)
  where
    buildP = \adr -> Person2{_name = n, _age = a, _addr = adr}
```

```haskell
street :: Lens' Address String

addr :: Lens' Person2 Address


p = Person2{_name = "Ivan", _age = 16,
            _addr = Address{_street = "blabla", _house = 4}}
```

Хотелось бы делать так:

```haskell
ps = p ^. addr . street



p2 = p & addr . street .~ "42"
```

Оператор композиции линз

```
street :: Lens' Address String

addr :: Lens' Person2 Address

(<.>) :: Lens' s a -> Lens' a b -> Lens' s b   --addr <.> street :: Lens' Person2 String

(l1 <.> l2) f s = let x = s ^. l1
                      f_x2 = l2 f x
                  in l1 (\_ -> f_x2) s
```

l1 :: forall f . Functor f => (a -> f a) -> s -> f s

l2 :: forall f . Functor f => (b -> f b) -> a -> f a

```
(<.>) :: Lens' s a -> Lens' a b -> Lens' s b   --g :: b -> f b
(l1 <.> l2) g s = let x = s ^. l1
                      x2 = l2 g x              --f - некоторый функтор
                  in l1 (const x2) s
                                               --x2 :: f a
```

Попробуем упростить

l1 :: forall f . Functor f => (a -> f a) -> s -> f s

l2 :: forall f . Functor f => (b -> f b) -> a -> f a

g :: b -> f b, f - некоторый функтор

l2 g :: a -> f a

l1 (l2 g) :: s -> f s
l1 (l2 g) s :: f s — Это и есть наша композиция!

```
(<.>) :: Lens' s a -> Lens' a b -> Lens' s b
(l1 <.> l2) g s = l1 (l2 g) s
```

```
(<.>) :: Lens' s a -> Lens' a b -> Lens' s b
(l1 <.> l2) g = l1 $ l2 g
```

```
(<.>) :: Lens' s a -> Lens' a b -> Lens' s b
(l1 <.> l2) = l1 . l2
```

Стандартный оператор композиции — то, что нужно!

Немного примеров

```
p = Person2{_name = "Ivan", _age = 16,
_addr = Address{_street = "blabla", _house = 4}}
```

```
ps = p ^. addr . street
--"blabla"
```

```
p2 = p & addr . street .~ "42"
--Person2 {_name = "Ivan", _age = 16, _addr =
            Address {_street = "42", _house = 4}}
```

```
p3 = p & addr . street %~ (++ " 42")
--Person2 {_name = "Ivan", _age = 16, _addr =
            Address {_street = "blabla 42", _house = 4}}
```

Снова новый тип линзы!

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

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

                    Lens' s a = Lens s s a a


view :: Lens s t a b -> s -> a


set :: Lens s t a b -> b -> s -> t


over :: Lens s t a b -> (a -> b) -> s -> t
```

Пример

```
data Person = Person {name :: String, age :: Int}
deriving (Show)

data Robot = Robot {code :: Int, age_r :: Int} deriving
(Show)



nameRobotL :: Lens Person Robot String Int
nameRobotL f Person{name = n, age = a} = buildR <$> f n
  where
    buildR = \c -> Robot {code = c, age_r = a}
```

```
pn = p ^. nameRobotL
--"Ivan"
```

```
r = p & nameRobotL .~ 15
--Robot {code = 4, age_r = 15}
```

```
r2 = p & nameRobotL %~ length
--Robot {code = 4, age_r = 15}
```

```
f :: String -> Maybe Int
f "Ivan" = Just 42
f _ = Nothing

g :: String -> Maybe Int
g "Ivan" = Nothing
g _ = Just 42
```

```
r3 = p & nameRobotL f
-- Just (Robot {code = 42, age_r = 15})
```

```
r4 = p & nameRobotL g
--Nothing
```

Полезные ресурсы

https://hackage.haskell.org/package/lens - страница пакета lens на hackage

https://github.com/ekmett/lens - репозиторий на github

https://www.youtube.com/watch?v=cefnmjtAolY - видео презентации разработчика lens (опасно для неподготовленных слушателей!)

https://skillsmatter.com/skillscasts/4251-lenses-compositional-data-access-and-manipulation
- выступление Simon Peyton Jones (SPJ) с рассказом о линзах (для просмотра потребуется регистрация на сайте)

Некоторые из многочисленных туториалов

https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial

https://artyom.me/lens-over-tea-1