

Функциональное программирование

Лекция 15. Зипперы и линзы

Денис Николаевич Москвин

СПбАУ РАН

19.12.2017

- 1 Зипперы
- 2 Линзы
- 3 Призмы, траверсы и т.п.
- 4 Пользовательские линзы

- 1 Зипперы
- 2 Линзы
- 3 Призмы, траверсы и т.п.
- 4 Пользовательские линзы

- В чистых функциональных языках внесение изменений в существующую структуру может быть не очень эффективным (например, последний элемент списка).
- Чтобы добраться до нужного узла, требуется снять много конструкторов, а затем «надеть» их обратно.

```
update2 :: new -> (a, (old, c)) -> (a, (new, c))  
update2 v (x, (_, z)) = (x, (v, z))
```

Сессия GHCi

```
GHCi> stru = (1, (2, (3, 4)))  
GHCi> update2 42 stru  
(1, (42, (3, 4)))  
GHCi> update2 "Hello" stru  
(1, ("Hello", (3, 4)))
```

Понятие зиппера

- Идея (Gerard Huet, 1997): смонтировать структуру, похожую на исходную, но обеспечивающую:
 - возможность навигации по структуре;
 - эффективную модификацию элемента в текущем месте (фокусе, hole) внутри структуры.

```
type Triple a = (a, (a, a))

type TripleZ a = (a,          -- фокус
                  Cntx a) -- контекст

data Cntx a = C1 ((), (a, a)) | C2 (a, ((), a)) | C3 (a, (a, ()))
```

- Естественно в деле лучше использовать изоморфное

```
data Cntx a = C1 a a | C2 a a | C3 a a
```

Зиппер для списка

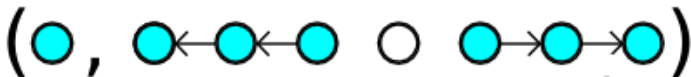
```
type ListZ a = (a, CntxL a)
type CntxL a = ([a],[a])

mklz :: [a] -> ListZ a
mklz (x:xs) = (x,([],xs))

fwd :: ListZ a -> ListZ a
fwd (e,(xs,y:ys)) = (y,(e:xs,ys))
```

```
GHCi> lz = mklz [0..3]
(0,([],[1,2,3]))
GHCi> fwd lz
(1,([0],[2,3]))
GHCi> (fwd . fwd) lz
(2,([1,0],[3]))
```

Зиппер для списка (2)



```
bwd :: ListZ a -> ListZ a
bwd (e,(x:xs,ys)) = (x,(xs,e:ys))
```

```
unlz :: ListZ a -> [a]
unlz (x,([],xs)) = x:xs
unlz z = unlz (bwd z)
```

```
GHCi> lz' = (fwd . fwd) lz
(2,([1,0],[3]))
GHCi> bwd lz'
(1,([0],[2,3]))
GHCi> unlz lz'
[0,1,2,3]
```

Зиппер для списка: изменения в фокусе

Внесение изменений в значение в фокусе:

```
updLZ :: a -> ListZ a -> ListZ a  
updLZ v (_,ctx) = (v,ctx)
```

```
insLZ :: a -> ListZ a -> ListZ a  
insLZ v (e,(xs,ys)) = (v,(xs,e:ys))
```

```
delLZ :: ListZ a -> ListZ a  
delLZ (e,(xs,y:ys)) = (y,(xs,ys))  
delLZ (e,(x:xs,[])) = (x,(xs,[]))
```

```
GHCi> (unlz . updLZ 42 . fwd . fwd . mklz) [0..3]  
[0,1,42,3]
```

```
GHCi> (unlz . insLZ 33 . insLZ 42 . fwd . fwd . mklz) [0..3]  
[0,1,33,42,2,3]
```



```
type Triple a = (a,(a,a))  
  
type TripleZ a = (a, Cntx a)  
  
data Cntx a = C1 a a | C2 a a | C3 a a
```

Переводя на язык теории типов:

```
Triple X  $\cong$  X3  
  
Cntx X  $\cong$  3 * X2
```

Верно ли очевидное наблюдение в общем случае?

```
type Triple a = (a,(a,a))  
  
type TripleZ a = (a, Cntx a)  
  
data Cntx a = C1 a a | C2 a a | C3 a a
```

Переводя на язык теории типов:

```
Triple X  $\cong$  X3  
  
Cntx X  $\cong$  3 * X2
```

Верно ли очевидное наблюдение в общем случае? Да!

Рекурсивный тип списка

Вспомним определения списка с прошлой лекции

$$L(X) = 1 + X + X^2 + X^3 + \dots$$

Мы переписывали его в виде рекурсивного уравнения:

$$L(X) = 1 + X * (1 + X + X^2 + X^3 + \dots)$$

$$L(X) = 1 + X * L(X)$$

Можно использовать еще более экстремистский подход

$$L(X) - X * L(X) = 1$$

$$L(X) * (1 - X) = 1$$

$$L(X) = 1 / (1 - X)$$

Сравним первое и последнее, вспомнив ряды.

Продифференцируем

$$L(X) = 1 / (1 - X)$$

$$L'(X) = 1 / (1 - X)^2$$

$$L'(X) = L(X) * L(X)$$

Сравним с zipperом:

```
type ListZ a = (a, CntxL a)
```

```
type CntxL a = ([a], [a])
```

Контекст с дыркой определяется производной параметризованного типа по его параметру.

Зиппер для двоичного дерева

```
data Tree a = Empty | Node (Tree a) a (Tree a)

type TreeZ a = (a, CntxT a)
```

Найдем контекст дифференцированием:

$$T(X) = 1 + X * T^2(X)$$

$$T'(X) = T^2(X) + X * 2 * T(X) * T'(X)$$

$$T'(X) * (1 - 2 * X * T(X)) = T^2(X)$$

$$T'(X) = T^2(X) / (1 - 2 * X * T(X))$$

$$T'(X) = T^2(X) * L(2 * X * T(X))$$

$$T'(X) = T(X) * T(X) * L(2 * X * T(X))$$

- 1 Зипперы
- 2 Линзы**
- 3 Призмы, траверсы и т.п.
- 4 Пользовательские линзы

- Линза — инструмент для манипулирования подструктурой некоторой структуры данных.
- Линзы доступны, например, через модуль `Control.Lens` библиотеки `lens`.
- Например, `_1` и `_2` — линзы для доступа к первому и второму элементам пары:

```
GHCi> view _1 (7,8)
7
GHCi> (7,8) ^. _2
8
```


- Композиция линз — это линза:

```
GHCi> view (_1 . _2) ((7,8),9)
8
```

Обратите внимание на обратный порядок при композиции!

- Оператор $(\sim.)$ (инфиксный эквивалент `view`) обеспечивает доступ к полям в OO-стиле:

```
GHCi> ((7,8),9) ~. _1
(7,8)
GHCi> ((7,8),9) ~. _1 . _2
8
```

Модификация с помощью линз

- Линзы позволяют модифицировать подструктуру в фокусе:

```
GHCi> set _1 42 (7,8)
(42,8)
GHCi> set _1 "Hello" (7,8)
("Hello",8)
GHCi> over _1 (^2) (7,8)
(49,8)
```

- У `set` и `over` есть инфиксные эквиваленты:

```
GHCi> _1 .~ "Hello" $ (7,8)
("Hello",8)
GHCi> _1 %~ (^2) $ (7,8)
(49,8)
GHCi> (7, 8) & _1 .~ "Hello"
("Hello",8)
```

- Линза — инструмент для манипулирования элементом типа `a` некоторой структуры данных типа `s`, находящимся в фокусе этой линзы.
- Линза — это АТД составленный из пары геттер-сеттер

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

- Линзы должны удовлетворять следующим законам:

```
view l (set l v s)  ≡  v
set l (view l s) s  ≡  s
set l v' (set l v s) ≡  set l v' s
```

Реализация линз: наивный подход

```
data LensNaive s a = MkLens (s -> a) (s -> a -> s)

_1Naive :: LensNaive (a,b) a
_1Naive = MkLens (\(x,_) -> x) (\(_,y) v -> (v,y))

viewNaive :: LensNaive s a -> s -> a
viewNaive (MkLens get _) s = get s
```

```
*SimpleLens> viewNaive _1Naive (5,7)
5
```

- неэффективно (конструктор данных MkLens дает дополнительный барьер во время исполнения);
- имеет проблемы с расширением и обобщением (например, хотелось бы, чтобы композиция линз была линзой).

Линзы ван Лаарховена (Functor transformer lenses)

Линза — это функция, которая превращает вложение a в функтор f во вложение s в этот функтор.

```
{-# LANGUAGE Rank2Types #-}  
type Lens s a = forall f. Functor f =>  
(a -> f a) -> s -> f s
```

Как упаковать в такую конструкцию геттер и сеттер?

```
--      (s -> a) -> (s -> a -> s) -> (a -> f a) -> s -> f s  
lens :: (s -> a) -> (s -> a -> s) -> Lens s a  
lens get set = \ret s -> fmap (set s) (ret $ get s)
```

Пример для пар

```
_1 :: Lens (a,b) a    -- (a -> f a) -> (a,b) -> f (a,b)  
_1 = lens (\(x,_) -> x) (\(_,y) v -> (v,y))
```

Как вынуть из линзы геттер и сеттер?

Как вынуть из линзы геттер и сеттер? Использовать вместо `f` подходящий функтор!

Как вынуть из линзы геттер и сеттер? Использовать вместо `f` подходящий функтор! Для геттера это фантомный функтор:

```
newtype Const a x = Const { getConst :: a }
-- Const      :: a -> Const a x
-- getConst :: Const a x -> a
instance Functor (Const a) where
  fmap _ (Const v) = Const v
```


Как вынуть из линзы геттер и сеттер? Использовать вместо `f` подходящий функтор! Для геттера это фантомный функтор:

```
newtype Const a x = Const { getConst :: a }
-- Const      :: a -> Const a x
-- getConst :: Const a x -> a
instance Functor (Const a) where
  fmap _ (Const v) = Const v
```

```
--      ((a -> Const a a) -> s -> Const a s) -> s -> a
view :: Lens s a                               -> s -> a
view lns s = getConst (lns Const s)
```

```
*SimpleLens> view (_2 . _1) (5,(6,7))
6
```

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

```
instance Functor Identity where  
  fmap f (Identity x) = Identity (f x)
```

```
-- ((a->Identity a)->s->Identity s) -> (a -> a) -> s -> s  
over :: Lens s a          -> (a -> a) -> s -> s  
over lns fn s = runIdentity $ lns (Identity . fn) s  
-- over lns fn s = set s (fn (get s)) -- set и get из lns
```

```
*SimpleLens> over _1 (+5) (6,7)  
(11,7)
```

```
-- ((a->Identity a)->s->Identity s) -> (a -> a) -> s -> s
over :: Lens s a                -> (a -> a) -> s -> s
over lns fn s = runIdentity $ lns (Identity . fn) s
```

```
--      ((a->Identity a)->s->Identity s) -> a -> s -> s
set :: Lens s a                -> a -> s -> s
set lns a s = over lns (const a) s
--set lns a s = runIdentity $ lns (Identity . const a) s
```

```
*SimpleLens> set _2 42 (5,7)
(5,42)
*SimpleLens> set (_2 . _1) 33 ("abc", (6,True))
("abc", (33,True))
```

- 1 Зипперы
- 2 Линзы
- 3 Призмы, траверсы и т.п.**
- 4 Пользовательские линзы

- Призмы это инструмент двойственный к линзам. Они используются для типов сумм, как линзы — для типов произведений.
- Призма выбирает одну из ветвей типа суммы или терпит неудачу. Например, `_Left :: Prism' (Either a b) a`

```
GHCi> preview _Left (Left "Hello")
Just "Hello"
GHCi> preview _Right (Left "Hello")
Nothing
GHCi> review _Left "Hello"
Left "Hello"
```

```
preview, (^?) :: Prism' s a -> s -> Maybe a
review :: Prism' s a -> a -> s
```

```
_Cons :: Prism' [a] (a, [a])
```

```
GHCi> [1,2,3] ^? _Cons  
Just (1,[2,3])  
GHCi> [] ^? _Cons  
Nothing
```

- Призмы и линзы допускают взаимную композицию.

```
GHCi> Left (7,8,9) ^? _Left . _2
Just 8
GHCi> (Left 7,Left 8,Right "Hello") ^? _3 . _Right
Just "Hello"
GHCi> (Left 7,Left 8,Right "Hello") ^? _3 . _Left
Nothing
```

- композиция линз — линза, композиция призм — призма.
- Другие (перекрестные) композиции относятся к типу `Traversal` — конструкции, которая может иметь ноль, один или много фокусов.

Пример траверса для списка

- `_tail` — траверс, держащий в фокусе весь хвост списка:

```
GHCi> :i _tail
_tail :: Cons s s a a => Traversal' s s
      -- Defined in 'Control.Lens.Cons'
GHCi> [1,2,3] ^. _tail
[2,3]
GHCi> [1,2,3] & _tail .~ [4,5,6,7]
[1,4,5,6,7]
GHCi> import Data.Sequence as Seq
GHCi> fromList [1,2,3] ^. _tail
fromList [2,3]
```

- `_tail` универсален для стандартных контейнеров-последовательностей.

- 1 Зипперы
- 2 Линзы
- 3 Призмы, траверсы и т.п.
- 4 Пользовательские линзы

Template Haskell

- Template Haskell — расширение для типобезопасного метапрограммирования во время компиляции.

```
GHCi> :set -XTemplateHaskell
GHCi> import Language.Haskell.TH
```

- *Оксфордские скобки* [| |] позволяют получить AST из кода, типа, объявления или образца:

```
GHCi> let ast = runQ [| \x -> x |]
GHCi> ast
LamE [VarP x_0] (VarE x_0)
GHCi> runQ [t| IO Bool |]
AppT (ConT GHC.Types.IO) (ConT GHC.Types.Bool)
GHCi> runQ [d| f x = 42 |]
[FunD f_1 [Clause [VarP x_2] (NormalB (LitE (IntegerL 42))) []]]
```

- Специальный синтаксис $\$(\dots)$ позволяет генерировать код по AST:

```
GHCi> let ast = runQ [| \x -> x |]  
GHCi> :t $(ast)  
$(ast) :: t -> t  
GHCi> $(ast) 42  
42
```

- Ручное конструирование AST осуществляют в специальной монаде Q (от слова Quotation):

```
GHCi> let constNGen n = do {var <- newName "x";  
return $ LamE [VarP var] (LitE (IntegerL n))}  
GHCi> :t constNGen  
constNGenf :: Integer -> Q Exp  
GHCi> $(constNGen 42) "Answer?"  
42
```

- Зададим типы данных для широты и долготы, выраженных в градусах, минутах и секундах:

```
data Arc = Arc {
  _degree, _minute, _second :: Int
} deriving Show
data Location = Location {
  _latitude , _longitude :: Arc
} deriving Show
```

- Символ подчеркивания в именах полей записи является конвенцией, принятой в `Control.Lens` для генерации линз с помощью `TH`.
- Можно было бы конструировать линзы вручную, без `TH`, но это утомительное, хотя и несложное занятие.

- Вызовы TH

```
$(makeLenses ''Location)  
$(makeLenses ''Arc)
```

создадут линзы с именами полей без подчеркивания:
latitude :: Lens' Location Arc и т.д.

- Теперь можно использовать их как геттеры и сеттеры:

```
GHCi> auLocation = Location (Arc 60 0 9) (Arc 30 22 26)  
GHCi> auLocation ^. latitude . degree  
60  
GHCi> auLocation & longitude . second .~ 27  
Location {_latitude = Arc {_degree = 60, _minute = 0, _second = 9},  
_longitude = Arc {_degree = 30, _minute = 22, _second = 27}}
```