# Функциональное программирование Лекция 14. Чисто функциональные структуры данных

Денис Николаевич Москвин

СПбАУ РАН

21.05.2015

# План лекции

1. Зипперы
2. Алгебра и анализ зипперов
3. Линзы и призмы
4. Пользовательские линзы

# План лекции

# Проблемы неизменяемости

- В чистых функциональных языках внесение изменений в существующую структуру данных часто неэффективно.
- Например, вставка значения в середину списка требует его «пересборки».

```
insertAfter5 :: Int -> [Int] -> [Int]
insertAfter5 v xs  = helper [] xs where
  helper _ [] = []
  helper hs (e : ts)
    | e == 5    = hs ++ e : v : ts
    | otherwise = helper (hs ++ [e]) ts
```

## Сессия GHCi

```
> insertAfter5 42 [1..10]
[1,2,3,4,5,42,6,7,8,9,10]
```

# 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.
- Например, зиппер для списка представляет собой пару списков

```
type LZ a =
  (
   [a] -- хвост
  ,[a] -- контекст
  )
```

# Зиппер для списка: создание

Вкладываем список в зиппер:

```
mklz :: [a] -> LZ a
mklz xs = (xs, [])
```

## Сессия GHCi

```
> let lzp = mklz [0..3]
> lzp
([0,1,2,3],[])
```

Вынимаем список из зиппера:

```
unlz :: LZ a -> [a]
unlz (xs, []) = xs
unlz (xs, y:ys) = unlz (y:xs, ys)
```

# Зиппер для списка: навигация

Навигация вперёд — контекст нарастает, хвост облезает:

```
fwd :: LZ a -> LZ a
fwd (x:xs, ys) = (xs, x:ys)
```

## Сессия GHCi

```
> fwd lzp
([1,2,3],[0])
> (fwd . fwd . fwd) lzp
([3],[2,1,0])
```

Навигация назад — контекст сокращается, хвост нарастает:

```
bck :: LZ a -> LZ a
bck (xs, y:ys) = (y:xs, ys)
```

# Зиппер для списка: изменение в фокусе

Внесение изменений в значение в фокусе:

```
insertLZ :: a -> LZ a -> LZ a
insertLZ v (xs, ys) = (v:xs, ys)

deleteLZ :: LZ a -> LZ a
deleteLZ (_:xs, ys) = (xs, ys)

updateLZ :: a -> LZ a -> LZ a
updateLZ v (_:xs, ys) = (v:xs, ys)
```

## Сессия GHCi

```
> (unlz . updateLZ 42 . fwd . fwd) lzp
[0,1,42,3]
```

# Зиппер для двоичного дерева

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

type TZ a = (Tree a, -- хвост
             Ctx a)  -- контекст
```

Контекст либо пуст (`Top`), либо содержит информацию о «направлении на родителя» (`L` или `R`) вместе со значением родителя (`a`), другим поддеревом родителя (`Tree a`) и, рекурсивно, контексте родительского узла (`Ctx a`):

```
data Ctx a = Top
           | L (Ctx a) a (Tree a)
           | R (Tree a) a (Ctx a)
```

(Позже мы получим нерекурсивное определение контекста.)

# Зиппер для дерева: создание

Вкладываем дерево в зиппер и вынимаем обратно

```
mktz :: Tree a -> TZ a
mktz t = (t, Top)

untz :: TZ a -> Tree a
untz (t, Top) = t
untz z        = untz $ up z
```

## Сессия GHCi

```
> let tr = Node (Node Empty 1 Empty) 2 (Node (Node Empty 3
Empty) 4 (Node Empty 5 Empty))
> let tzp = mktz tr
> tzp
(Node (Node Empty 1 Empty) 2 (Node (Node Empty 3 Empty) 4
(Node Empty 5 Empty)), Top)
```

# Зиппер для дерева: навигация

```
left (Node l x r, c) = (l, L c x r)

right (Node l x r, c) = (r, R l x c)

up (t, L c x r) = (Node t x r, c)
up (t, R l x c) = (Node l x t, c)
```

## Сессия GHCi

```
> right tzp
(Node (Node Empty 3 Empty) 4 (Node Empty 5 Empty),
R (Node Empty 1 Empty) 2 Top)
> left $ right tzp
(Node Empty 3 Empty,
L (R (Node Empty 1 Empty) 2 Top) 4 (Node Empty 5 Empty))
> (up . up . left . right) tzp == tzp
True
```

# Зиппер для дерева: модификация

Заменяем значение дерева в фокусе:

```
updateTZ :: a -> TZ a -> TZ a
updateTZ v (Node l x r, c) = (Node l v r, c)
```

## Сессия GHCi

```
> tr
Node (Node Empty 1 Empty) 2 (Node (Node Empty 3 Empty) 4
(Node Empty 5 Empty))
> (untz . updateTZ 42 . left . right . mktz) tr
Node (Node Empty 1 Empty) 2 (Node (Node Empty 42 Empty) 4
(Node Empty 5 Empty))
```

# План лекции

# Рекурсивный тип списка

Вспомним определения списка с прошлой лекции

```
L(X) = 1 + X + X^2 + X^3 + ...
```

Мы переписывали его в виде рекурсивного уравнения:

```
L(X) = 1 + X * (1 + X + X^2 + X^3 + ...)
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.

# Производная от типа списка

Продифференцируем

```
L'(X) = 1 * L(X) + x * L'(X)
L'(X) (1 - X) = L(X)
L'(X) = L(X) / (1 - X)
L'(X) = L(X) * (1 + X + X^2 + X^3 + ...)
L'(X) = L(X) * L(X)
```

Сравним с зиппером:

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

Является ли это совпадением?

# Производная от типа списка

Продифференцируем

```
L'(X) = 1 * L(X) + x * L'(X)
L'(X) (1 - X) = L(X)
L'(X) = L(X) / (1 - X)
L'(X) = L(X) * (1 + X + X^2 + X^3 + ...)
L'(X) = L(X) * L(X)
```

Сравним с зиппером:

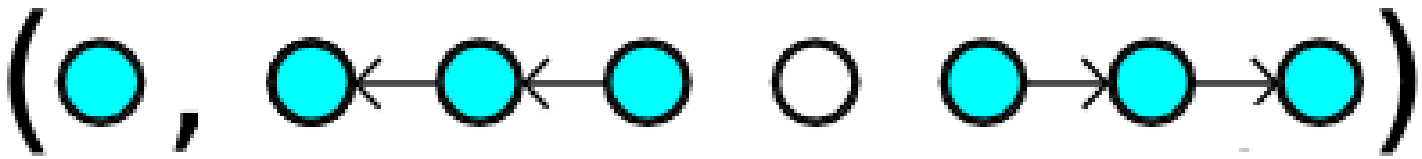
```
type LZ a = ([a],[a])
```

Является ли это совпадением? Нет!

# Контекст с дыркой (One-Hole Context)

- Если мы хотим добиться универсальности, нужно ввести понятие *контекста с дыркой*, вытащив из исходной структуры элемент и факторизовав оставшееся.
- Тогда зиппер представляет собой пару из элемента в фокусе и контекста с дыркой.
- В частности, для списка:

```
type ListZipper a = (a,          -- элемент
                     ([a],[a])) -- контекст
```



- Conor McBride заметил, что именно контекст с дыркой определяется производной параметризованного типа по его параметру.

# Упражнение

- Как выглядит контекст с дыркой для гомогенной пары?

# Упражнение

- Как выглядит контекст с дыркой для гомогенной пары?

```
(X^2)' = 2 * X = X + X
2 * X = (Bool, X)
X + X = Either X X

PairZipper a = (a, Either a a)
```

# Упражнение

- Как выглядит контекст с дыркой для гомогенной пары?

```
(X^2)' = 2 * X = X + X
2 * X = (Bool, X)
X + X = Either X X

PairZipper a = (a, Either a a)
```

- гомогенной тройки?

# Упражнение

- Как выглядит контекст с дыркой для гомогенной пары?

```
(X^2)' = 2 * X = X + X
2 * X = (Bool, X)
X + X = Either X X

PairZipper a = (a, Either a a)
```

- гомогенной тройки?
- гетерогенной пары?
- гетерогенной тройки?

# Контекст с дыркой для бинарного дерева

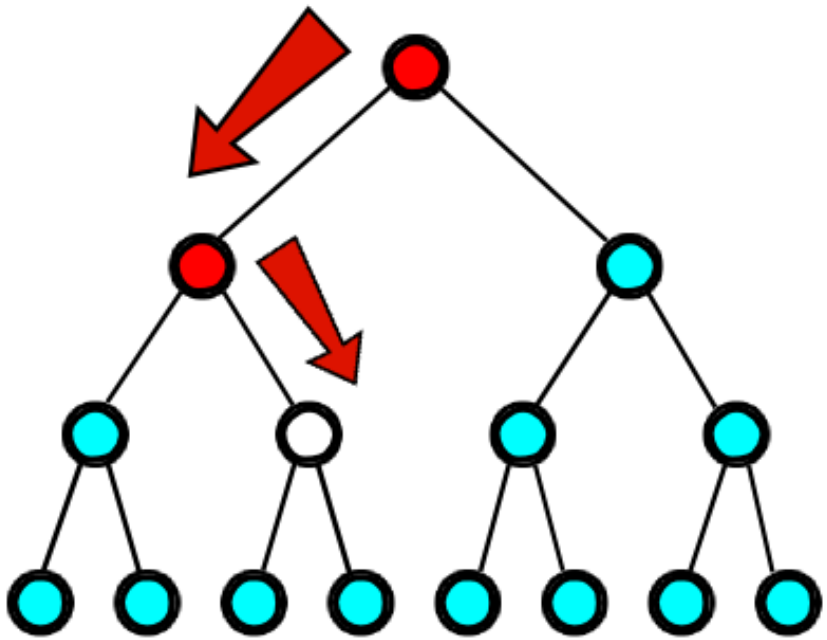
Бинарное дерево

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

порождает контекст с дыркой

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

# Контекст с дыркой для бинарного дерева (2)



`T'(X) = T(X) * T(X) * L(2 * X * T(X))` задает факторизацию:

`T(X) * T(X)` – два поддерева ниже фокуса;

`L(2 * X * T(x)) = [(Bool,x,Tree x)]`, где

`Bool` – указывает, идти вверх направо или налево;

`x` – значение родительского узла;

`Tree x` – второе поддерево родительского узла.

# План лекции

# Линзы

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

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

# Композиция линз

- Композиция линз — это линза:

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

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

```
> ((7,8),9) ^. _1
(7,8)
> ((7,8),9) ^. _1 . _2
8
```

# Модификация с помощью линз

- Линзы позволяют модифицировать подструктуру в фокусе:

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

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

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

# Призмы

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

```
> preview _Left (Left "Hello")
Just "Hello"
> preview _Right (Left "Hello")
Nothing
*Fp14lens> review _Left "Hello"
Left "Hello"
```

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

# Призмы для списков

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

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

# Композиции призм и линз

Призмы и линзы допускают взаимную композицию.

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

# План лекции

# Template Haskell

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

```
> :set -XTemplateHaskell
> :m + Language.Haskell.TH
```

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

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

# Template Haskell: код из AST

- Специальный синтаксис `$(...)` позволяет генерировать код по AST:

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

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

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

# Пользовательские линзы

- 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.

# Пользовательские линзы

- 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:

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