

Функциональное программирование

Лекция 5. Программирование на языке Haskell

Денис Николаевич Москвин

СПбАУ, CSC

12.03.2015

- 1 Ленивость и строгость
- 2 Алгебраические типы данных и сопоставление с образцом
- 3 Списки и работа с ними

- 1 Ленивость и строгость
- 2 Алгебраические типы данных и сопоставление с образцом
- 3 Списки и работа с ними

Сколько значений у типа Bool?

- Всякое выражение в Haskell имеет значение определенного типа.
- Сколько значений у типа Bool?
- На первый взгляд два — True и False, в соответствии с определением:

```
data Bool = True | False
```

Сколько значений у типа Bool?

- Всякое выражение в Haskell имеет значение определенного типа.
- Сколько значений у типа Bool?
- На первый взгляд два — True и False, в соответствии с определением:

```
data Bool = True | False
```

- Но это не так!

Значение незавершающегося вычисления

- Рассмотрим выражение `bot :: Bool`, определённое рекурсивно

```
bot = not bot
```

- Его значение — не `True` и не `False`, а \perp (основание). В Haskell'e \perp — значение, разделяемое всеми типами:

```
 $\perp :: \text{forall } a. a$ 
```

- Ошибкам (но не исключениям!) тоже приписывается это значение.

- Haskell гарантирует вызов-по-необходимости (таково поведение по умолчанию)

```
const42 x = 42
```

```
Prelude> const42 bot  
42
```

- Такие функции как `const42`, игнорирующие значение своего аргумента, называются *нестрогими* по этому аргументу.
- Для *строгих* функций, наоборот, всегда выполняется

$$f \perp = \perp$$

- Для форсированного вычисления значения используют специальный комбинатор $\text{seq} :: a \rightarrow b \rightarrow b$

$$\text{seq } \perp b = \perp$$

$$\text{seq } a b = b, \text{ если } a \neq \perp$$

- С чисто синтаксической точки зрения seq это $\lambda x y \rightarrow y$.
- Но он «нарушает» ленивую семантику языка, позволяя форсировать вычисление без необходимости!

Как сильно `seq` форсирует?

- `seq` «потворствует» распространению \perp , интересуясь значением своего первого аргумента

```
Prelude> seq undefined 42
*** Exception: Prelude.undefined
Prelude> seq (id undefined) 42
*** Exception: Prelude.undefined
```

- Однако конструкторы данных и лямбда-абстракции, являясь «значениями», обеспечивают барьер для распространения \perp

```
Prelude> seq (undefined,undefined) 42
42
Prelude> seq (\x -> undefined) 42
42
```

- Через `seq` определяется энергичная аппликация (с вызовом-по-значению)

```
infixr 0 $!  
($!) :: (a -> b) -> a -> b  
f $! x = x 'seq' f x
```

- Форсирование приводит к «худшей определенности»

GHCi

```
Prelude> const42 undefined  
42  
Prelude> const42 $! undefined  
*** Exception: Prelude.undefined
```

- Вспомним факториал с аккумулярующим параметром

```
factorial n = helper 1 n
  where helper acc k | k > 1 = helper (acc * k) (k - 1)
                  | otherwise = acc
```

- Из-за ленивости acc будет содержать thunk
(((1 * n) * (n - 1)) * (n - 2) ...)
- Оптимизатор GHC обычно справляется, но можно, не полагаясь на него, написать

```
factorial n = helper 1 n
  where helper acc k | k > 1 = (helper $! acc * k) (k - 1)
                  | otherwise = acc
```

- 1 Ленивость и строгость
- 2 Алгебраические типы данных и сопоставление с образцом
- 3 Списки и работа с ними

Функция, переставляющую элементы пары

```
swap      :: (a,b) -> (b,a)
swap (x,y) = (y,x)
```

Выражение (x,y) представляет собой *образец*. При вызове

```
*Fp05> swap (5,True)
(True,5)
```

происходит *сопоставление с образцом*:

- проверяется, что конструктор $(,)$ — подходящий;
- переменные x и y связываются со значениями 5 и True .

Алгебраические типы данных: перечисления

- Перечисление — тип с 0-арными конструкторами данных

```
data Color = Red | Green | Blue | Indigo | Violet
  deriving Show
```

```
*Fp05> :type Red
Red :: Color
```

- Сопоставление с образцом происходит сверху вниз

```
isRGB      :: Color -> Bool
isRGB Red  = True
isRGB Green = True
isRGB Blue  = True
isRGB _    = False      -- Wild-card
```

Алгебраические типы данных: декартово произведение

Тип-произведение с одним конструктором данных

```
data PointDouble = PtD Double Double deriving Show
```

```
*Fp05> :type PtD  
PtD :: Double -> Double -> PointDouble
```

```
midPointDouble :: PointDouble -> PointDouble -> PointDouble  
midPointDouble (PtD x1 y1) (PtD x2 y2) =  
  PtD ((x1 + x2) / 2) ((y1 + y2) / 2)
```

```
*Fp05> midPointDouble (PtD 3.0 5.0) (PtD 9.0 8.0)  
PtD 6.0 6.5
```

Полиморфные типы

- Тип точки можно параметризовать типовым параметром:

```
data Point a = Pt a a deriving Show
```

```
*Fp05> :type Pt  
Pt :: a -> a -> Point a
```

- `Point` — оператор над типами, конкретный тип получается его аппликацией к некоторому типу, например, `Int`.

```
*Fp05> :kind Point  
Point :: * -> *  
*Fp05> :kind Point Int  
Point Int :: *
```


«Избыточный» полиморфизм и умолчания (defaulting)

```
midPoint :: Fractional a => Point a -> Point a -> Point a
midPoint (Pt x1 y1) (Pt x2 y2) =
  Pt ((x1 + x2) / 2) ((y1 + y2) / 2)
```

```
*Fp05> :type midPoint (Pt 3 5) (Pt 9 8)
midPoint (Pt 3 5) (Pt 9 8) :: Fractional a => Point a
*Fp05> midPoint (Pt 3 5) (Pt 9 8)
Pt 6.0 6.5
*Fp05> :type it
it :: Point Double
```

- Но (+) и (/) определены только для конкретных типов — контекст `Fractional a` задаёт *ad hoc полиморфизм*.
- По умолчанию подразумевается, что для любого модуля `default (Integer, Double)`

```
data List a = Nil | Cons a (List a) deriving Show
```

- Конструкторы имеют тип `Nil :: List a` и `Cons :: a -> List a -> List a`.
- Обработка — через рекурсию и сопоставление с образцом

```
len      :: List a -> Int
len Nil  = 0
len (Cons _ xs) = 1 + len xs
```

```
*Fp05> let myList = Cons 'a' (Cons 'b' (Cons 'c' Nil))
*Fp05> len myList
3
```

- Встроены, но могли бы быть определены так

```
data [] a = [] | a : ([] a)
infixr 5 :
```

- Для удобства введён синтаксический сахар

```
[1,2,3] == 1:2:3:[]
```

Пример определения функции

```
head      :: [a] -> a
head (x:_) = x
head []    = error "Prelude.head: empty list"
```

```
head (x:_)    = x
head []       = error "head: empty list"
```

ЭКВИВАЛЕНТНО

```
head' xs = case xs of
  (x:_) -> x
  []     -> error "head': empty list"
```

- Поскольку `case ... of ...` — выражение, его можно использовать в любом месте кода.
- Если нас интересует только один случай, можно использовать образец в лямбде

```
head'' = \ (x:_) -> x
```

Семантика сопоставления с образцом

- Сопоставление с образцом происходит сверху-вниз, затем слева-направо.
- Сопоставление с образцом может быть
 - успешным (succeed);
 - неудачным (fail);
 - расходящимся (diverge).

```
foo (1, 2) = 3  
foo (0, _) = 5
```

- (0, undefined) — неудача в первом, успех во втором;
- (undefined, 0) — расходимость в первом же образце;
- (2, 1) — две неудачи и, как следствие, расходимость;
- (1, 5-3) — ???

В определении функции

```
dupFirst      :: [a] -> [a]
dupFirst (x:xs) = x:x:xs
```

мы можем присвоить псевдоним всему образцу, используя затем этот псевдоним в правой части определения

```
dupFirst'      :: [a] -> [a]
dupFirst' s@(x:xs) = x:s
```

Неопровержимые (irrefutable) образцы

- К неопровержимым относятся wild-cards (`_`), as-образцы, формальные параметры-переменные и *ленивые образцы*.
- Тильда задаёт *ленивый образец*: сопоставление с ним всегда проходит успешно, а связывание откладывается до момента использования

```
(***) f g ~(x, y) = (f x, g y)
```

GHCi

```
*Fp05> (const 1 *** const 2) undefined  
(1,2)
```

- **Строгий конструктор данных.** Флаг строгости (!) в конструкторе данных позволяет форсировать вычисление соответствующего поля

```
data Complex a = !a :+: !a
infix 6 :+:
```

- **Bang pattern.** Позволяет форсировать вычисление при связывании в образцах. Является расширением GHC.

```
Prelude> :set -XBangPatterns
Prelude> let foo !x = True
Prelude> foo undefined
*** Exception: Prelude.undefined
```


- Ключевое слово `type` задаёт *синоним типа*:

```
type String = [Char]
```

- Для удобства введён синтаксический сахар

```
"Hello" == ['H','e','l','l','o']
```

- Ключевое слово `newtype` задаёт *новый тип* с единственным конструктором, упаковывающий уже существующий тип:

```
type Age1 = Int  
newtype Age2 = Age Int
```

Метки полей (Field Labels)

- Для доступа к полям типа-произведения, например, `data Point a = Pt a a`, приходится использовать специальные селекторы `\(Pt x _) -> x` или `\(Pt _ y) -> y`. Можно при определении типа дать полям метки, облегчающие такой доступ

```
data Point' a = Pt' {ptx, pty :: a} deriving Show
```

- Метки имеют тип `Point' a -> a` и работают как селекторы

```
*Fp05> let myPt = Pt' 3 2
*Fp05> ptx myPt
3
```

```
*Fp05> let myPt2 = Pt' {ptx = 3}

<interactive>:1:13:
  Warning: Fields of 'Pt'' not initialised: pty
  In the expression: Pt' {ptx = 3}
  In an equation for 'myPt2': myPt2 = Pt' {ptx = 3}
*Fp05> :t myPt2
myPt2 :: Point' Integer
*Fp05> ptx myPt2
3
*Fp05> pty myPt2
*** Exception: <...>:1:13-25:
  Missing field in record construction Fp05.pty
*Fp05> let myPt3 = Pt' {ptx = 3, pty = 2}
```

Использование меток полей

- Можно использовать метки полей как селекторы

```
absP p = sqrt (ptx p ^ 2 + pty p ^ 2)
```

- Можно связать их с переменными в образце

```
absP' (Pt' {ptx = x, pty = y}) = sqrt (x ^ 2 + y ^ 2)
```

- С помощью меток полей структуры можно «обновлять»

```
*Fp05> let myPt4 = Pt' {ptx = 7, pty = 8}
*Fp05> myPt4
Pt' {ptx = 7, pty = 8}
*Fp05> myPt4 {ptx = 42}
Pt' {ptx = 42, pty = 8}
```

Стандартные алгебраические типы

- Тип `Maybe a` позволяет задать «необязательное» значение

```
data Maybe a = Nothing | Just a
maybe :: b -> (a -> b) -> Maybe a -> b

find :: (a -> Bool) -> [a] -> Maybe a
```

- Тип `Either a b` описывает одно значение из двух

```
data Either a b = Left a | Right b
either :: (a -> c) -> (b -> c) -> Either a b -> c

head'''' :: [a] -> Either String a
head'''' (x:_) = Right x
head'''' [] = Left "head''': empty list"
```

- 1 Ленивость и строгость
- 2 Алгебраические типы данных и сопоставление с образцом
- 3 Списки и работа с ними

```
tail      :: [a] -> [a]
tail (_:xs) = xs
tail []    = error "tail: empty list"
```

```
take      :: Int -> [a] -> [a]
take n _  | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
(++)      :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

Какова сложность tail? конкатенации?

Функции высших порядков (HOF)

Первый аргумент — унарный предикат:

```
filter      :: (a -> Bool) -> [a] -> [a]
filter upred [] = []
filter upred (x:xs)
  | upred x    = x : filter upred xs
  | otherwise  = filter upred xs
```

Первый аргумент — произвольная функция:

```
map        :: (a -> b) -> [a] -> [b]
map _ []   = []
map f (x:xs) = f x : map f xs
```



```
length      :: [a] -> Int
length []   = 0
length (_:xs) = 1 + length xs
```

- Реализация `length` в GHC

```
length      :: [a] -> Int
length l    = len l 0#
  where
    len      :: [a] -> Int# -> Int
    len []   a# = I# a#
    len (_:xs) a# = len xs (a# +# 1#)
```

- # маркирует *unboxed types*.
- Рекурсия в `len` — хвостовая.

«Бесконечные» структуры данных

Рекурсия позволяет описывать «бесконечные» структуры данных:

GHCi

```
Prelude> let ones = 1 : ones
Prelude> :type ones
ones :: [Integer]
```

Благодаря ленивости вычисляется только то, что требуется:

GHCi

```
Prelude> let numsFrom n = n : numsFrom (n+1)
Prelude> let squares = map (^2) (numsFrom 0)
Prelude> take 10 squares
[0,1,4,9,16,25,36,49,64,81]
```

Имеется компактный способ описывать большие «регулярные» списки:

GHCi

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> [1,3..17]
[1,3,5,7,9,11,13,15,17]
Prelude> ['A'..'z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz"
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
...]
```

Для формирования «нелинейных» последовательностей есть другая техника...

Выделение списков (List Comprehension)

GHCi

```
Prelude> [x^2 | x <- [0..9]]  
[0,1,4,9,16,25,36,49,64,81]
```

При нескольких генераторах чаще обновляется тот, что правее:

GHCi

```
Prelude> [(x,y,z) | x<-[1..19], y<-[1..19], z<-[1..19],  
x^2+y^2==z^2]  
[(3,4,5), (4,3,5), (5,12,13), (6,8,10), (8,6,10), (8,15,17),  
(9,12,15), (12,5,13), (12,9,15), (15,8,17)]
```