Курс: Функциональное программирование

Лекция 5. Программирование на языке Haskell

Денис Николаевич Москвин

21.10.2011

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

План лекции

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

План лекции

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

Сколько значений у типа `Bool`?

Всякое выражение в Haskell имеет значение определенного типа.

Сколько разных значений у выражений 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'е $\perp$ — значение, разделяемое всеми типами:

$$\perp :: \texttt{forall a. a}$$

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

Ленивая семантика

Haskell гарантирует вызов-по-необходимости (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

$$\mathtt{f}\ \bot\ =\ \bot$$

Как форсировать вычисления

Для форсированного вычисления значения используют специальнный комбинатор `seq :: a -> b -> b`

$$\begin{aligned} \texttt{seq}\ \bot\ \texttt{b} &= \bot \\ \texttt{seq a b} &= \texttt{b},\ \text{если}\ \texttt{a} \neq \bot \end{aligned}$$

С чисто синтаксической точки зрения `seq` это `\x y -> 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
```

Аппликация с вызовом по значению

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»

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

Вспомним факториал с аккумулятором

Аккумулирующий параметр использовался для экономии памяти под кадры стека

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

Однако из-за ленивости `acc` может содержать цепочку thunk'ов

```
(((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
```

План лекции

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

Сопоставление с образцом (pattern matching)

Рассмотрим функцию, 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
```

Конструкторы данных имеют тип `Color`:

```
*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

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

Конструктор данных имеет тип функции

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

Пример использования

```
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 двумерной точки может быть параметризован ти́повым параметром `a`:

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

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

`Point` — оператор над типами, конкретные типы получаются аппликацией к определенному типу, например `Float`.

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

Умолчания (defaulting)

Функции над полиморфными типами — полиморфны:

```
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 над конкретными типами — контекст `Fractional a` задаёт *ad hoc полиморфизм*.
По умолчанию подразумевается, что для любого модуля
`default (Integer, Double)`

Рекурсивные типы

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

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

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

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

Стандартные списки

Могли бы быть определены так (на самом деле — встроены)

```
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"
```

Выражение `case ... of ...`

Определение функции

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

эквивалентно следующей форме

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

Семантика сопоставления с образцом

Сопоставление с образцом происходит сверху-вниз, затем слева-направо. Сопоставление с образцом может быть

- успешным (succeed);
- неудачным (fail);
- расходящимся (diverge).

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

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

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)

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

Форсирование строгости

Строгие конструкторы 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
```

As-образец

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
```

Объявления `type` и `newtype`

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

```
data Point a = Pt a a
```

приходится использовать специальные селекторы типа `\(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 имееют тип `Point' a -> a` и работают как селекторы

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

Инициализация в синтаксисе с метками полей

```
*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)
```

Но можно связать метки полей 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` представляет «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` представляет 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"
```

План лекции

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

Основные функции из `Data.List` (1)

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

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

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

Основные функции из Data.List (2)

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

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

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

Основные функции из `Data.List` (3)

Реализация 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

```
Prelude> let ones = 1 : ones
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]
```

Арифметические последовательности

```
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,
...
```

Выделение списков

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