Функциональное программирование
Лекция 5. Типы данных

Денис Николаевич Москвин

ИТМО, магистратура JB SE

05.10.2020

План лекции

1. Ленивость и строгость
2. Алгебраические типы данных и сопоставление с образцом
3. Стандартные списки и работа с ними
4. Образцы: дополнительные сведения
5. Типы данных: дополнительные сведения

План лекции

1. Ленивость и строгость
2. Алгебраические типы данных и сопоставление с образцом
3. Стандартные списки и работа с ними
4. Образцы: дополнительные сведения
5. Типы данных: дополнительные сведения

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

Вспомним выражение `bot :: Bool`, определённое рекурсивно

```
bot :: Bool
bot = not bot
```

Его значение — не `True` и не `False`, а $\bot$ (основание, дно). В Haskell'е $\bot$ — значение, разделяемое всеми типами:

```
⊥ :: a
```

Ошибкам тоже приписывается это значение.

```
GHCi> :t undefined
undefined :: a
GHCi> :t error
error :: [Char] -> a
```

Нестрогая (ленивая) семантика

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

```
ignore x  = 42
```

```
GHCi> ignore undefined
42
GHCi> ignore bot
42
```

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

```
f ⊥ = ⊥
```

Как форсировать вычисления

- Для форсированного вычисления используют функцию

```
seq :: a -> b -> b
seq ⊥ b = ⊥
seq a b = b, если a ≠ ⊥
```

- Синтаксически `seq` похожа на `\a b -> b`. Но она нарушает ленивую семантику языка, позволяя форсировать вычисление без необходимости.
- `seq` потворствует распространению $\bot$, интересуясь значением своего первого аргумента

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

Как сильно seq форсирует?

- seq производит вычисление своего первого аргумента, если в нем имеется редекс на верхнем уровне.
- Однако конструкторы данных, лямбда-абстракции и частично примененные функции, являясь «значениями», обеспечивают барьер для распространения $\perp$

```
GHCi> seq (undefined,undefined) 42
42
GHCi> seq (\x -> undefined) 42
42
GHCi> seq ((+) undefined) 42
42
```

- Подобные «не редексы» объединяют одним термином – их называют *слабой головной нормальной формой* (weak head normal form, WHNF).

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»

```
GHCi> ignore undefined
42
GHCi> ignore $! undefined
*** Exception: Prelude.undefined
```

Пример использования seq

- Вспомним факториал с аккумулирующим 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 вида
  (...((1 * n) * (n - 1)) * (n - 2) * ... * 2)
- Оптимизатор GHC обычно справляется, имея встроенный *анализатор строгости*. 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

1. Ленивость и строгость
2. Алгебраические типы данных и сопоставление с образцом
3. Стандартные списки и работа с ними
4. Образцы: дополнительные сведения
5. Типы данных: дополнительные сведения

Сопоставление с образцом (pattern matching)

Функция, переставляющую элементы пары

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

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

```
GHCi> swap (5+2,True)
(True,7)
```

происходит *сопоставление с образцом*:

- проверяется, что конструктор `(,)` — подходящий (для пары это тривиально);
- переменные `x` и `y` связываются со выражениями `5+2` и `True`;
- осуществляется подстановка выражений вместо переменных в теле функции `swap`.

Алгебраические типы данных: тип суммы

Перечисление — тип с 0-арными конструкторами данных

```
data CardinalDirection = North | East | South | West
```

Конструкторы данных имеют тип `CardinalDirection`:

```
GHCi> dir = North
GHCi> :t dir
dir :: CardinalDirection
GHCi> dir
 error: No instance for (Show CardinalDirection)
        arising from a use of `print'
    In a stmt of an interactive GHCi command: print it
```

Исправить можно так:

```
data CardinalDirection = North | East | South | West
    deriving Show
```

Перечисления: сопоставление с образцом

```
data CardinalDirection = North | East | South | West
    deriving Show
```

Сопоставление с образцом происходит сверху вниз

```
hasPole :: CardinalDirection -> Bool
hasPole North = True
hasPole South = True
hasPole _     = False
```

Подчеркивание (или переменная) задают *неопровержимый* образец.

```
GHCi> hasPole North
True
GHCi> hasPole West
False
```

Встроенные типы перечислений

Встроенные типы данных ведут себя так, как будто они определены как перечисления

```
data Char = '\NUL' | ... | 'a' | 'b' | 'c' | 'd' | ...
          | '\1114111'
data Int = -9223372036854775808 | ...
         | -2 | -1 | 0 | 1 | 2 | ...
         | 9223372036854775807
data Integer = ... | -2 | -1 | 0 | 1 | 2 | ...
```

Это позволяет использовать соответствующие литералы как образцы

```
isAnswer :: Integer -> Bool
isAnswer 42 = True
isAnswer _  = False
```

Семантика сопоставления с образцом

- Сопоставление происходит сверху вниз, затем слева направо.
- Сопоставление бывает
  - успешным (succeed);
  - неудачным (fail);
  - расходящимся (diverge).

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

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

Алгебраические типы данных: декартово произведение

Тип-произведение с одним конструктором данных

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

```
GHCi> :type PtD
PtD :: Double -> Double -> PointDouble
```

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

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

Полиморфные типы

Тип точки можно параметризовать ти́повым параметром:

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

```
GHCi> :type Pt
Pt :: a -> a -> Point a
```

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

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

Кайнды — система типов над системой типов Haskell.

Полиморфные функции над полиморфными типами

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

```
GHCi> :type midPoint (Pt 3 5) (Pt 9 8)
midPoint (Pt 3 5) (Pt 9 8) :: Fractional a => Point a
GHCi> midPoint (Pt 3 5) (Pt 9 8)
Pt 6.0 6.5
```

- Полиморфные типы полиморфны параметрически, то есть на типовый параметр невозможно наложить ограничения. (В старых версиях GHC и по стандарту можно!)
- Но (+) и (/) определены только для конкретных типов — контекст `Fractional a` задаёт ***ad hoc полиморфизм***.

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"
```

Экспоненциальные типы

Экспоненциальный тип — это тип функции.

```
data Endom a = Endom (a -> a)

appEndom :: Endom a -> a -> a
appEndom (Endom f) = f
```

```
GHCi> e = Endom (\n -> 2 * n + 3)
GHCi> :t e
e :: Num a => Endom a
GHCi> :t appEndom e
appEndom e :: Num a => a -> a
GHCi> e `appEndom` 5
13
```

Рекурсивные типы

Допустимо в полях конструктора данных ссылаться на определяемый конструктор типа. Например, тип чисел Пеано:

```
GHCi> data Nat = Zero | Suc Nat deriving Show
GHCi> :t Zero
Zero :: Nat
GHCi> :t Suc
Suc :: Nat -> Nat
GHCi> two = Suc (Suc Zero)
GHCi> {pred (Suc n) = n; pred Zero = Zero}
GHCi> pred two
Suc Zero
```

Хотя этот тип структурно похож на числа Черча, вычисление предессора намного эффективнее, благодаря механизму сопоставления с образцом.

Рекурсивные типы: список

```
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
```

```
GHCi> myList = Cons 'a' (Cons 'b' (Cons 'c' Nil))
GHCi> len myList
3
```

План лекции

1. Ленивость и строгость
2. Алгебраические типы данных и сопоставление с образцом
3. Стандартные списки и работа с ними
4. Образцы: дополнительные сведения
5. Типы данных: дополнительные сведения

Стандартные списки

Встроены, но могли бы быть определены так

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

Для удобства введён синтаксический сахар

```
[1,2,3] ≡ 1:(2:(3:[])) ≡ 1:2:3:[]
```

Пример определения функции

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

Это частичная функция, в современном Haskell использовать их не рекомендуется.

Основные функции из Data.List

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

```
GHCi> tail [1,2,3,4]
[2,3,4]
GHCi> tail "ABCD"
"BCD"
```

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

Какова сложность `tail`? конкатенации?

Основные функции из Data.List (2)

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

```
GHCi> take 3 "ABCDEFG"
"ABC"
GHCi> take 10 "ABCDEFG"
"ABCDEFG"
```

```
drop                    :: Int -> [a] -> [a]
drop                    = undefined
```

Как через drop сделать тотальный эквивалент tail?

Функции высших порядков (HOF)

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

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

```
GHCi> map length ["Good","bye","world"]
[4,3,5]
GHCi> map (^2) . map length $ ["Good","bye","world"]
[16,9,25]
```

Семейства zip и zipWith

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs

zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]

unzip :: [(a,b)] -> ([a],[b])
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ []     _      = []
zipWith _ _      []     = []
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs

zipWith3 :: (a -> b -> c -> d)
         -> [a] -> [b] -> [c] -> [d]
```

«Бесконечные» структуры данных

«Бесконечные» структуры данных описываются рекурсией:

```
GHCi> ones = 1 : ones
GHCi> :type ones
ones :: Num a => [a]
```

Благодоря ленивости вычисляется только то, что требуется:

```
GHCi> numsFrom n = n : numsFrom (n+1)
GHCi> squares = map (^2) (numsFrom 0)
GHCi> take 10 squares
[0,1,4,9,16,25,36,49,64,81]
GHCi> fibs = 0 : 1 : zipWith (+) fibs (drop 1 fibs)
GHCi> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

Арифметические последовательности

Имеется компактный способ описывать большие «регулярные» списки:

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

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

Выделение списков (List Comprehension)

Название происходит из аксиоматической теории множеств.

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

При нескольких генераторах чаще обновляется тот, что правее:

```
GHCi> [ [x,y] | x <- "ABC", y <- "de" ]
["Ad","Ae","Bd","Be","Cd","Ce"]
```

Выделение списков: дополнительные возможности

Генераторы могут ссылаться на значения из предыдущих генераторов; можно использовать предикаты над этими значениями:

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

Можно использовать сопоставление с образцом:

```
GHCi> tst = [Just 2,Nothing,Just 3]
GHCi> [x | Just x <- tst]
[2,3]
```

План лекции

1. Ленивость и строгость
2. Алгебраические типы данных и сопоставление с образцом
3. Стандартные списки и работа с ними
4. Образцы: дополнительные сведения
5. Типы данных: дополнительные сведения

Выражение case ... of ...

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

транслируется в Kernel следующим образом

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

Общее правило трансляции

$$
\begin{array}{l}
\texttt{f } p_{11} \ \ldots\ p_{1k} = e_1 \\
\ldots \\
\texttt{f } p_{n1} \ \ldots\ p_{nk} = e_n
\end{array}
\quad\equiv\quad
\begin{array}{l}
\texttt{f } x_1 \ \ldots\ x_k = \texttt{case } (x_1,\ldots,x_k) \texttt{ of} \\
\quad (p_{11},\ldots,p_{1k}) \texttt{ -> } e_1 \\
\quad \ldots \\
\quad (p_{n1},\ldots,p_{nk}) \texttt{ -> } e_n
\end{array}
$$

Поскольку `case ... of ...` — выражение, его можно использовать в любом месте кода.

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' ys@(x:xs)  =  x:ys
```

Ленивые образцы

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

```
(***) :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)
(***) f g ~(x, y) = (f x, g y)
```

- Без лени в образце следующий код расходился бы

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

Образцы в let-выражениях

В `let`-выражениях можно использовать образцы:

```
GHCi> let x:xs = "ABC" in xs ++ xs
"BCBC"
```

Правило трансляции нерекурсивного `let` в Kernel

```
let p = e1 in e  ≡  case e1 of ~p -> e
-- see Haskell Report 3.12 for complete translation
```

Обратите внимание на маркер ленивости:

```
GHCi> let x:xs = [] in 42
42
GHCi> let x:xs = undefined in 42
42
GHCi> let x:xs = [] in x
*** Exception: Non-exhaustive patterns in x : xs
```

Образцы в лямбда-абстракциях

- В лямбда-абстракциях тоже можно использовать образцы

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

- Общее правило трансляции лямбды с образцами в Kernel

$$\backslash p_1\ \ldots\ p_n\ \texttt{->}\ e_1 \quad \equiv$$
$$\backslash x_1\ \ldots\ x_n\ \texttt{->}\ \texttt{case}\ (x_1,\ldots,x_n)\ \texttt{of}\ (p_1,\ldots,p_n)\ \texttt{->}\ e_1$$

  Здесь все x_i — свежие переменные.

- Недостаток: можно обработать только один образец на один аргумент лямбды.
- Имеется расширение `LambdaCase`, решающее эту проблему.

Охранные образцы (pattern guards)

В Haskell 2010 синтаксис охранных выражений был расширен

```
firstOdd :: [Integer] -> Integer
firstOdd xs | Just x <- find odd xs = x
            | otherwise             = 0

firstOddIsBig :: [Integer] -> Bool
firstOddIsBig xs
  | Just x <- find odd xs, x > 1000 = True
  | otherwise                       = False
```

```
GHCi> firstOdd [2,3,4]
3
GHCi> firstOddIsBig [2,3,4,1001]
False
GHCi> firstOddIsBig [2,4,1001]
True
```

План лекции

1. Ленивость и строгость
2. Алгебраические типы данных и сопоставление с образцом
3. Стандартные списки и работа с ними
4. Образцы: дополнительные сведения
5. Типы данных: дополнительные сведения

Синтаксис записей: 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 :: a, ptY :: a }
```

Метки имееют тип `Point a -> a` и работают как проекции

```
GHCi> myPt = Pt 3 2
GHCi> ptX myPt
3
```

Типы данных, поля которых снабжены метками, называют *записями* (records).

Инициализация в синтаксисе с метками полей

Порядок полей при инициализации произволен:

```
GHCi> myPt1 = Pt {ptY = 2, ptX = 3}
GHCi> myPt1
Pt {ptX = 3, ptY = 2}
```

Можно даже инициализировать не все поля ...

```
GHCi> myPt2 = Pt {ptX = 3}
 warning: [-Wmissing-fields] Fields of `Pt' not initial
ised: ptY
GHCi> ptX myPt2
3
GHCi> ptY myPt2
*** Exception: Missing field in record construction ptY
```

... но лучше этого не делать.

Использование меток полей

Стандартное использование в качестве проекций

```
absP p = sqrt (ptX p ^ 2 + ptY p ^ 2)
```

Можно связать метки полей с переменными в образце

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

Следующее выглядит лучше предыдущего, но это не всегда так. Догадайтесь в каких случаях более многословные метки лучше.

```
absP'' (Pt x y) = sqrt (x ^ 2 + y ^ 2)
```

С помощью меток полей записи можно «обновлять»

```
GHCi> myPt3 = Pt {ptX = 7, ptY = 8}
GHCi> myPt3 {ptX = 42}
Pt {ptX = 42, ptY = 8}
```

Общие метки полей

Метки полей одного типа могут быть общими в нескольких конструкторах данных:

```
data Homo = Known   {name :: String, male :: Bool}
          | Unknown {male :: Bool}
```

```
GHCi> john = Known "John" True
GHCi> stranger = Unknown False
GHCi> male john
True
GHCi> male stranger
False
```

Одинаковые метки полей для *разных типов* недопустимы, их область видимости — глобальная. Добавив `data Bad = Bad {male :: Bool}`, получим ошибку компиляции: Multiple declarations of 'male'.

Объявление type

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

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

- Синонимы типа могут быть параметризованными:

```
GHCi> type EC = Either Char
GHCi> :kind EC
EC :: * -> *
GHCi> type LEC b = [EC b]
GHCi> le = [Right 5, Left 'z'] :: LEC Int
GHCi> :t le
le :: LEC Int
```

Объявление newtype

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

```
newtype AgeNT = AgeNT Int
data    AgeDT = AgeDT Int
fNT (AgeNT n) = 42
fDT (AgeDT n) = 42
```

При компиляции обертка `newtype` убирается, помимо эффективности исполнения это приводит к лучшей определенности:

```
GHCi> fNT undefined
42
GHCi> fDT undefined
*** Exception: Prelude.undefined
```

Фантомные типы

Фантомные типы позволяют на уровне типов хранить дополнительную информацию, используемую при проверке.

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Temperature a = Temperature Double
  deriving (Num,Eq,Show)

data Celsius
data Fahrenheit

comfortTemperature :: Temperature Celsius
comfortTemperature = Temperature 23

c2f :: Temperature Celsius -> Temperature Fahrenheit
c2f (Temperature c) = Temperature (1.8 * c + 32)
```

Фантомные типы: использование

Теперь типы гарантируют, что арифметические операции допустимы только «внутри» каждой температуры.

```
GHCi> :t comfortTemperature
comfortTemperature :: Temperature Celsius
GHCi> comfortTemperature + Temperature 2
Temperature 25.0
GHCi> c2f comfortTemperature
Temperature 73.4
GHCi> :t c2f comfortTemperature
c2f comfortTemperature :: Temperature Fahrenheit
GHCi> c2f comfortTemperature - comfortTemperature
 error: Couldn't match type `Celsius' with `Fahrenheit'
```

Подключенное расширение `GeneralizedNewtypeDeriving` позволяет автоматически генерировать представителя класса типов `Num`.

Форсирование строгости

Флаг строгости ! в конструкторе данных позволяет форсировать вычисление соответствующего поля

```
infix 6 :+
data Complex a = !a :+ !a        -- Data.Complex
```

Сравним поведение пары и комплексного числа

```
GHCi> case (1,undefined) of (_,_) -> 42
42
GHCi> case 1 :+ undefined of _ :+ _ -> 42
*** Exception: Prelude.undefined
```

При этом вычисление полей форсируется, только когда форсируется вычисление родительской структуры, поэтому

```
GHCi> case 1 :+ undefined of _ -> 42
42
```