Функциональное программирование
Лекция 5'. Типы данных

Денис Николаевич Москвин

НИУ ВШЭ — СПб, ШИФТ,
бакалавриат ПМИ, 2 курс

07.10.2025

План лекции

1. Алгебраические типы данных и сопоставление с образцом
2. Образцы: дополнительные сведения
3. Типы данных: дополнительные сведения

План лекции

Сопоставление с образцом (pattern matching)

Функция над парой, игнорирующая свой аргумент

```
ignore :: Num p => (a, b) -> p
ignore (x,y) = 42
```

Сопоставление с образцом происходит энергично

```
GHCi> ignore undefined
*** Exception: Prelude.undefined
```

Сопоставление с образцом (pattern matching)

Функция над парой, игнорирующая свой аргумент

```
ignore :: Num p => (a, b) -> p
ignore (x,y) = 42
```

Сопоставление с образцом происходит энергично

```
GHCi> ignore undefined
*** Exception: Prelude.undefined
```

Однако вычисление форсируется только до WHNF, в данном случае — конструктора пары

```
GHCi> ignore (undefined,undefined)
42
```

Алгебраические типы данных: тип суммы, перечисление

```
data TC = DC1 | ... | DCk,        k ⩾ 0
```

```
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) ...
```

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
```

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

Алгебраические типы данных: декартово произведение

$$\texttt{data}\ \mathrm{TC} = \mathrm{DC}\ \mathrm{TC}_1\ \ldots\ \mathrm{TC}_k, \quad k \geqslant 0$$

```
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` — функция 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

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

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

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

```
\p1 ... pn -> e1   ≡
  \x1 ... xn -> case (x1,...,xn) of (p1,...,pn) -> e1
```

  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

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
ignoreNT (AgeNT n) = 42
ignoreDT (AgeDT n) = 42
```

При компиляции обертка `newtype` убирается. Помимо эффективности исполнения это приводит к лучшей определенности:

```
GHCi> ignoreNT undefined
42
GHCi> ignoreDT undefined
*** Exception: Prelude.undefined
```

Форсирование строгости и инфиксые конструкторы

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
```