CS314. Функциональное программирование

Лекция 4. Алгебраические типы данных

В. Н. Брагилевский

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

10 сентября 2016 г.

Содержание

Содержание

Синонимы типов

```haskell
evalSmth :: (Double, Double) -> (Double, Double) -> Double
evalSmth = ...
```

```haskell
type Point = (Double, Double)

evalSmth :: Point -> Point -> Double
evalSmth = ...
```

```
ghci> evalSmth (0,0) (3,4)
5.0
```

Тип String как синоним

```
ghci> :info String
type String = [Char]    -- Defined in ‘GHC.Base’
```

Содержание

Содержание

Примеры типов-перечислений

- Дни недели, месяцы.
- Логические значения.
- Пол (мужской/женский).
- Масть и значение в картах.
- Жанр.
- ...

Тип для пола (мужской—женский)

```haskell
data Sex = Male | Female
```

```haskell
pensionAge :: Sex -> Int
pensionAge Male = 60
pensionAge Female = 55
```

```
ghci> pensionAge Male
60
ghci> pensionAge Female
55
ghci> map pensionAge [Male, Male, Female, Male]
[60,60,55,60]
```

Типы и конструкторы значений

```
data Sex = Male | Female
```

- Sex — тип.
- Male, Female — конструкторы значений.

Проблема: обработка значений

```
ghci> Male
    No instance for (Show Sex)
      arising from a use of ‘print’
    Possible fix: add an instance declaration for (Show Sex)
    In a stmt of an interactive GHCi command: print it
```

```
ghci> Female == Female
    No instance for (Eq Sex)
      arising from a use of ‘==’
    Possible fix: add an instance declaration for (Eq Sex)
    In the expression: Female == Female
    In an equation for ‘it’: it = Female == Female
```

Автоматическое порождение экземпляров классов

```
data Sex = Male | Female deriving (Show, Eq)
```

```
ghci> Male
Male
ghci> [Male, Male, Female, Male]
[Male, Male, Female, Male]
ghci> Female == Female
True
```

Пример: игральные карты

Типы

```
data Suit = Spades | Clubs | Diamonds | Hearts
    deriving (Show, Eq, Ord)

data Value = Seven | Eight | Nine | Ten
           | Jack | Queen | King | Ace
    deriving (Show, Eq, Ord)

type Card = (Value, Suit)
type Pack = [Card]
```

Значения

```
mpack :: Pack
mpack = [(Eight, Clubs), (Ten, Spades), (Ace, Diamonds)]
```

Сравнение значений

```
ghci> Nine < Ace
True
ghci> (Nine, Spades) < (Ace, Clubs)
True
ghci> (Queen, Spades) < (Queen, Hearts)
True
```

Примеры функций

```
isRed :: Suit -> Bool
isRed Diamonds = True
isRed Hearts = True
isRed _ = False
```

```
isPicture :: Value -> Bool
isPicture v
    | v >= Jack = True
    | otherwise = False
```

Пример использования

```
ghci> mpack
[(Eight, Clubs), (Ten, Spades), (Ace, Diamonds)]
ghci> all (isPicture.fst) mpack
False
ghci> any (isRed.snd) mpack
True
```

Стандартный тип Ordering

```
data Ordering = LT | EQ | GT

compare :: Ord a => a -> a-> Ordering
```

Для сравнения: в языке C используются значения -1, 0, 1.

```
ghci> compare 5 8
LT
ghci> compare "abc" "aba"
GT
ghci> compare  (Queen, Spades) (Queen, Spades)
EQ
```

Функция Data.Ord.comparing

Тип

```
comparing :: Ord a => (b -> a) -> b -> b -> Ordering
```

Пример

```
ghci> sort [(1,'x'), (2, 'c'), (3, 'a')]
[(1,'x'),(2,'c'),(3,'a')]
ghci> :m +Data.List Data.Ord
ghci> :t sortBy
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
ghci> sortBy (comparing snd) [(1,'x'), (2, 'c'), (3, 'a')]
[(3,'a'),(2,'c'),(1,'x')]
ghci> maximumBy (comparing snd) [(1,'x'), (2, 'c'), (3, 'a')]
(1,'x')
```

Содержание

Тип с персональной информацией

```
type Name = String
type Age = Int
data Person = Person Name Age
     deriving (Show)

name :: Person -> String
name (Person nm _) = nm
```

```
ghci> let p1 = Person "Фредди Крюгер" 43
ghci> p1
Person "Фредди Крюгер" 43
ghci> name p1
"Фредди Крюгер"
```

Синтаксис записей

```
data Person = Person {
    firstName :: String
    , lastName :: String
    , age :: Int
    , height :: Float
    , phoneNumber :: String} deriving (Show)
```

```
ghci> :t height
height :: Person -> Float
ghci> :t firstName
firstName :: Person -> String
```

Использование

```
ghci> let p = Person {firstName="Фредди", lastName="Крюгер",
                      age=43, height=190, phoneNumber="22223232"}
ghci> phoneNumber p
"22223232"
ghci> firstName p
"Фредди"
ghci> let p2 = p {age=44}
ghci> age p2
44
```

Сопоставление с образцом vs функции-аксессоры

```
process :: Person -> ...
process (Person n a) = ... n ... a ...
```

```
process :: Person -> ...
process p = ... name p ... age p ...
```

- А что если структура значения типа поменяется, например, добавится новое поле?

Пример: индекс массы тела

```
data Person = Person {name :: String, age :: Int,
                      weight :: Double, height :: Double}

bmi1, bmi2, bmi3 :: Person -> Double
bmi1 p = weight p / (height p)^2
bmi2 (Person name age weight height) = weight / height ^ 2
bmi3 Person {weight=weight, height=height}
                                    = weight / height ^ 2
```

```
{-# LANGUAGE  NamedFieldPuns #-}
bmi4 Person{weight, height} = weight / height ^ 2
```

```
{-# LANGUAGE RecordWildCards #-}
bmi5 Person{..} = weight / height ^ 2
```

RecordWildCards

```
person :: Person
person = let
           name = "Иванов"
           age = 20
           weight = 80
           height = 185
         in Person{..}
```

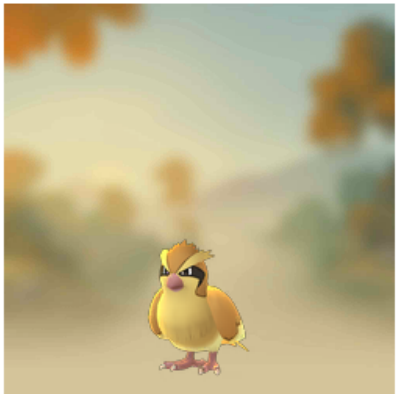
Пример: список покемонов

Постановка задачи

Дана строка следующего вида с информацией о покемонах и их CP:

```
Hypno 152
Eevee 300
Pidgey 10
```

Разработать АТД для хранения информации о покемоне и преобразовать заданную строку в список значений этого типа.



Решение

```
data Pokemon = Pokemon String Int
       deriving Show
```

```
parsePokemon :: String -> Pokemon
parsePokemon s = case words s of
                   [ty, cp] -> Pokemon ty (read cp)
                   _ -> error "неверный формат"
```

Функция read

```
ghci> :t read
read :: Read a => String -> a
```

```
parseInfo :: String -> [Pokemon]
parseInfo = map parsePokemon . lines
```

Проверка

```
ghci> parseInfo "Hypno 152\nEevee 300\nPidgey 10"
[Pokemon "Hypno" 152,Pokemon "Eevee" 300,Pokemon "Pidgey" 10]
```

Пример: поиск покемона с максимальным CP

```haskell
import Data.List
import Data.Ord

-- ...

maxCP :: [Pokemon] -> Pokemon
maxCP = maximumBy (comparing $ \ (Pokemon _ cp) -> cp)
```

Содержание

Пример: геометрическая фигура

```
data Point = Point Double Double
                deriving (Show)

type Radius = Double

data Shape = Circle Point Radius | Rectangle Point Point
                deriving (Show)
```

Пример: геометрическая фигура

```
area :: Shape -> Double
area (Circle _ r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2)) =
                              (abs $ x2 - x1) * (abs $ y2 - y1)
```

```
ghci> area (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> area (Circle (Point 0 0) 24)
1809.5574
```

Почему они алгебраические???

- Типы-перечисления — типы-суммы ($x \in T_1 + T_2$).
- Типы-контейнеры — типы-произведения ($x \in T_1 \times T_2$).
- Общий случай — сумма произведений.

Содержание

Тип Maybe a

Функция Data.List.find

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

Возможные результаты: либо элемент найден, либо нет.

```
ghci> find (<0) [5, 10, -3, 3, -4]
Just (-3)
ghci> find (<0) [5, 10, 3]
Nothing
```

```
data Maybe a = Nothing | Just a
```

Инструкция case

```
elem' a xs = case find (==a) xs of
                    Just _ -> True
                    Nothing -> False
```

Значения типа Maybe a

```
ghci> :t Nothing
Nothing :: Maybe a
ghci> :t Just 3
Just 3 :: Num a => Maybe a
ghci> :t Just 'a'
Just 'a' :: Maybe Char
```

Тип, конструктор типа, конструктор значения

```
data Maybe a = Nothing | Just a
```

- Maybe Integer — тип.
- Maybe — конструктор типа (функция, аргументом и значением которой являются типы).
- Just, Nothing — конструкторы значений.

Аналогия между Maybe a и [a]

- Nothing — пустой список [ ].
- Just 5 — одноэлементный список [5].

Некоторые функции из модуля Data.Maybe

```
maybe :: b -> (a -> b) -> Maybe a -> b
isJust :: Maybe a -> Bool
isNothing :: Maybe a -> Bool
fromJust :: Maybe a -> a
fromMaybe :: a -> Maybe a -> a
listToMaybe :: [a] -> Maybe a
maybeToList :: Maybe a -> [a]
catMaybes :: [Maybe a] -> [a]
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

- По типу функции можно понять о ней всё или почти всё!

Применение Maybe: функция Data.List.unfoldr

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

```
arith a d n = unfoldr next (a, 0)
  where
    next (a, k)
         | k < n = Just (a, (a+d, k+1))
         | otherwise = Nothing
```

```
fibs = 0:1:unfoldr next (0,1)
  where
    next (f1, f2) = let f3 = f1 + f2 in Just (f3, (f2, f3))
```

Тип Either a b

```
data Either a b = Left a | Right b
```

```
ghci> Right 20
Right 20
ghci> Left "xxx"
Left "xxx"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

Обычное применение: Right (результат) или Left (Ошибка).

Пример: параметризованные векторы

```
data Vector a = Vector a a a deriving (Show)

vplus :: (Num a) => Vector a -> Vector a -> Vector a
(Vector i j k) `vplus` (Vector l m n) =
                              Vector (i+l) (j+m) (k+n)

scalarProd :: (Num a) => Vector a -> Vector a -> a
(Vector i j k) `scalarProd` (Vector l m n) = i*l + j*m + k*n

vmult :: (Num a) => a -> Vector a -> Vector a
m `vmult` (Vector i j k) = Vector (m*i) (m*j) (m*k)
```

Пример: параметризованные векторы

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vmult` 10
Vector 30 90 70
ghci> Vector 2 9 3 `vmult`
                    (Vector 4 9 5 `scalarProd` Vector 9 2 4)
Vector 148 666 222
ghci> Vector 4 9 5 `scalarProd` Vector 9.0 2.0 4.0
74.0
```

Содержание

Примеры

- Список — это либо пустой список, либо голова и хвост (тоже список).
- Бинарное дерево — это либо пустое дерево, либо узел с двумя поддеревьями.
- Арифметическое выражение — это либо число, либо сумма выражений, либо произведение выражений.
- Рекурсивные типы часто параметризованы.

Список

```haskell
data List a = Nil | Cons a (List a)
```

```haskell
Cons 1 (Cons 2 (Cons 3 Nil)) :: Num a => List a
```

Извлечение первого элемента

```haskell
head' :: List a -> a
head' Nil = error "no such element"
head' (Cons a _) = a
```

Бинарное дерево

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
    deriving (Show)
```

```
Node 'g' (Node 'a' EmptyTree EmptyTree) EmptyTree
                                          :: Tree Char
```

Вставка в бинарное дерево поиска

Создание одноэлементного дерева

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree
```

Вставка элемента

```
treeInsert :: (Ord a, Eq a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right) =
    case compare x a of
      EQ -> Node a left right
      LT -> Node a (treeInsert x left) right
      GT -> Node a left (treeInsert x right)
```

Поиск элемента в БДП

```
treeElem :: (Ord a, Eq a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right) =
  case compare x a of
    EQ -> True
    LT -> treeElem x left
    GT -> treeElem x right
```

Формирование БДП

```
ghci> let nTree = foldr treeInsert EmptyTree [8,6,4,1,7,3,5]
ghci> nTree
Node 5
   (Node 3
     (Node 1 EmptyTree EmptyTree)
     (Node 4 EmptyTree EmptyTree)
   )
   (Node 7
     (Node 6 EmptyTree EmptyTree)
     (Node 8 EmptyTree EmptyTree)
   )
```

- Тип `treeInsert` *случайно* подошёл для `foldr`.

Представление арифметических выражений

Тип

```
data IntExpr = I Int                    -- целочисленная константа
             | Add IntExpr IntExpr      -- сумма двух выражений
             | Mul IntExpr IntExpr      -- произведение выражений
```

```
(I 5 `Add` I 1) `Mul` I 7 :: IntExpr
```

Вычисление

```
eval :: IntExpr -> Int
eval (I n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Содержание