

CS212. Парадигмы и технологии программирования: часть 1, функциональное программирование

Лекция 5. Классы типов в Haskell

В. Н. Брагилевский

14 марта 2019 г.

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

- Тип определяет множество возможных значений.
- Класс типов содержит сигнатуры функций, применимых к значениям некоторого типа (интерфейс).
- Экземпляр класса типов – это реализация функций из класса типов для конкретного типа.
- Говорят, что тип имеет экземпляр некоторого класса или что тип является частью класса.

Объявление класса

```
class ИмяКласса типоваяПеременная where  
    сигнатуры функций  
    реализации по умолчанию
```

Определение экземпляра

```
instance ИмяКласса Тип where  
    реализации функций
```

Классы типов Eq и Show

Класс типов Eq

Класс Eq определяет операции проверки на равенство (==) и неравенство (/=) для значений типа.

```
class Eq a where  
  (==) :: a -> a -> Bool  
  (/=) :: a -> a -> Bool  
  x == y = not (x /= y)  
  x /= y = not (x == y)
```

- Имена и типы функций
- Реализация по умолчанию
- Взаимно рекурсивная реализация (достаточно реализовать только одну из двух операций)

Экземпляр класса Eq для собственного типа

```
data TrafficLight = Red | Yellow | Green
```

```
instance Eq TrafficLight where
```

```
  Red == Red = True
```

```
  Green == Green = True
```

```
  Yellow == Yellow = True
```

```
  _ == _ = False
```

```
ghci> Red == Red
```

```
True
```

```
ghci> Red == Yellow
```

```
False
```

```
ghci> Red `elem` [Red, Yellow, Green]
```

```
True
```

В точности такой же экземпляр можно породить автоматически с использованием механизма deriving:

```
data TrafficLight = Red | Yellow | Green  
  deriving (Eq)
```

Экземпляр класса Eq для Maybe a

```
instance Eq a => Eq (Maybe a) where  
  Just x == Just y = x == y  
  Nothing == Nothing = True  
  _ == _ = False
```

- Стандартный экземпляр
- Ограничение на тип параметра

- Классы типов могут быть связаны отношением наследования (inheritance).

```
class Eq a => Ord a where  
  . . .
```

Экземпляр класса Show: светофор

```
data TrafficLight = Red | Yellow | Green
```

```
instance Show TrafficLight where
```

```
  show Red = "Red color"
```

```
  show Yellow = "Yellow color"
```

```
  show Green = "Green color"
```

```
ghci> [Red, Yellow, Green]
```

```
[Red color, Yellow color, Green color]
```

```
data Complex = Complex Double Double
```

```
instance Show Complex where
```

```
  show (Complex re im) = show re ++ " + i*"
                          ++ show im
```

```
ghci> Complex 4 6
```

```
4.0 + i*6.0
```

```
ghci> Complex 4.1 6.2
```

```
4.1 + i*6.2
```

Собственный класс типов «Да – Нет»

```
var a = if (0) "ДА!" else "НЕТ!"  
var b = if ("") "ДА!" else "НЕТ!"  
var c = if (false) "ДА!" else "НЕТ!"  
  
var a1 = if (1024) "ДА!" else "НЕТ!"  
var b1 = if ("КУ") "ДА!" else "НЕТ!"  
var c1 = if (true) "ДА!" else "НЕТ!"
```

Реализовать подобное поведение в языке Haskell невозможно из-за требований строгой типизации, но можно написать функцию-преобразователь к Bool.

Класс типов «Да–Нет»

Объявление класса типов

```
class YesNo a where  
  yesno :: a -> Bool
```

```
instance YesNo Int where  
  yesno 0 = False  
  yesno _ = True
```

```
instance YesNo [a] where  
  yesno [] = False  
  yesno _ = True
```

```
instance YesNo (Maybe a)  
  where  
  yesno (Just _) = True  
  yesno Nothing = False
```

```
instance YesNo Bool where  
  yesno b = b
```

Примеры использования

```
ghci> yesno $ length []
```

```
False
```

```
ghci> yesno "!!!"
```

```
True
```

```
ghci> yesno ""
```

```
False
```

```
ghci> yesno []
```

```
False
```

```
ghci> yesno [0,0,0]
```

```
True
```

```
ghci> yesno $ Just 0
```

```
True
```

```
ghci> yesno True
```

```
True
```

Слаботипизированный аналог конструкции if/then/else

```
yesnoIf :: YesNo y => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
    if yesno yesnoVal
      then yesResult
      else noResult
```

```
ghci> yesnoIf [] "ДА!" "НЕТ!"
"НЕТ!"
```

```
ghci> yesnoIf [2,3,4] "ДА!" "НЕТ!"
"ДА!"
```

```
ghci> yesnoIf (Just 500) "ДА!" "НЕТ!"
"ДА!"
```

```
ghci> yesnoIf Nothing "ДА!" "НЕТ!"
"НЕТ!"
```

Полугруппы и моноиды

Основные определения

Определение

Полугруппа – это множество с заданной на нём ассоциативной бинарной операцией $(S, *)$.

Определение

Моноид – это полугруппа с нейтральным элементом $(M, *, e)$:

$$\forall x \in M \quad x * e = e * x = x.$$

Определение

Группа – это моноид $(G, *, e)$, в котором каждый элемент имеет обратный:

$$\forall x \in G \quad \exists y \in G \quad x * y = y * x = e.$$

Классы типов Semigroup и Monoid

Data.Semigroup

```
class Semigroup a where  
  (<>) :: a -> a -> a  
  sconcat :: NonEmpty a -> a  
  stimes :: Integral b => b -> a -> a
```

Data.Monoid

```
class Semigroup a => Monoid a where  
  mempty :: a  
  mconcat :: [a] -> a  
  mconcat = foldr (<>) mempty  
  mtimes :: Integral b => b -> a -> a
```

Числовые моноиды и объявление `newtype`

- Сложение и 0 как нейтральный элемент.
- Умножение и 1 как нейтральный элемент.

```
newtype Sum a = Sum { getSum :: a }
```

```
newtype Product a = Product { getProduct :: a }
```

- Ключевое слово **newtype** создаёт тип-обёртку для значений некоторого типа.
- У типа, определяемого с помощью **newtype**, может быть только один конструктор значения с единственным параметром оборачиваемого типа.

Экземпляры Semigroup и Monoid для типов Product и Sum

```
instance Num a => Semigroup (Sum a) where  
  Sum x <> Sum y = Sum (x + y)
```

```
instance Num a => Semigroup (Product a) where  
  Product x <> Product y = Product (x * y)
```

```
instance Num a => Monoid (Sum a) where  
  mempty = Sum 0
```

```
instance Num a => Monoid (Product a) where  
  mempty = Product 1
```

Примеры

```
ghci> getProduct $ Product 3 <> Product 9
27
ghci> getProduct $ Product 3 <> mempty
3
ghci> Product 3 <> Product 4 <> Product 2
Product {getProduct = 24}
ghci> getProduct . mconcat . map Product $ [3,4,2]
24
ghci> getSum $ Sum 2 <> Sum 9
11
ghci> getSum $ mempty <> Sum 3
3
ghci> getSum . mconcat . map Sum $ [1,2,3]
6
```

```
newtype Any = Any { getAny :: Bool }
```

```
instance Semigroup Any where
```

```
    Any x <> Any y = Any (x || y)
```

```
instance Monoid Any where
```

```
    mempty = Any False
```

- Аналогично объявляется тип **All** и экземпляры для него.

Примеры

```
ghci> getAny $ Any True <> Any False  
True
```

```
ghci> mconcat $ map Any [False, False, True]  
Any { getAny = True }
```

```
ghci> getAll $ mempty <> All True  
True
```

```
ghci> mconcat $ map All [True, True, False]  
All { getAll = False }
```

- Использование типа a как моноида.
- Предпочтение первого значения (обёртка First).
- Предпочтение последнего значения (обёртка Last).

Экземпляры для Maybe a

```
instance Semigroup a => Semigroup (Maybe a) where  
  Nothing <> m = m  
  m <> Nothing = m  
  Just m1 <> Just m2 = Just (m1 <> m2)
```

```
instance Monoid a => Monoid (Maybe a) where  
  mempty = Nothing
```

- Похожим образом объявляются экземпляры для **First** и **Last**.

```
ghci> Nothing <> Just "QQ"  
Just "QQ"  
ghci> Just (Sum 3) <> Just (Sum 4)  
Just (Sum {getSum = 7})  
ghci> Just (Product 3) <> Just (Product 4)  
Just (Product {getProduct = 12})
```

```
ghci> First (Just 'a') <> First (Just 'b')
```

```
First { getFirst = Just 'a' }
```

```
ghci> First Nothing <> First (Just 'b')
```

```
First { getFirst = Just 'b' }
```

```
ghci> mconcat $ map Last [Nothing,Just 9,Just 10]
```

```
Last { getLast = Just 10 }
```

```
ghci> Last (Just "one") <> Last (Just "two")
```

```
Last { getLast = Just "two" }
```

`mempty` $\langle \rangle$ `x` = `x`

`x` $\langle \rangle$ `mempty` = `x`

`(x` $\langle \rangle$ `y)` $\langle \rangle$ `z` = `x` $\langle \rangle$ `(y` $\langle \rangle$ `z)`

Код, написанный в расчёте на наличие экземпляра класса типов `Monad`, может работать с данными самых разных типов. Такой код называется обобщённым (`generic`).

Пример абстрактного процесса обработки

```
process :: (Eq m, Monoid m) =>  
          m -> m -> [m] -> m
```

```
process x y zs  
  | x<>y == mempty = mempty  
  | otherwise = mconcat zs
```

```
ghci> process "" "" ["1", "2", "3", "4"]  
""
```

```
ghci> process "12" "13" ["1", "2", "3", "4"]  
"1234"
```

```
ghci> process (Sum 5) (Sum (-5)) (map Sum [1..4])  
Sum {getSum = 0}
```

```
ghci> process (Sum 5) (Sum 1) (map Sum [1..4])  
Sum {getSum = 10}
```

Сорта типов

```
ghci> :kind Int
Int :: *
ghci> :k Char
Char :: *
```

Сорта конструкторов типов с параметрами

```
ghci> :k Maybe
```

```
Maybe :: * -> *
```

```
ghci> :k Maybe Int
```

```
Maybe Int :: *
```

```
ghci> :k Either
```

```
Either :: * -> * -> *
```

```
ghci> :k Either String
```

```
Either String :: * -> *
```

```
ghci> :k Either String Int
```

```
Either String Int :: *
```

```
ghci> :k []
```

```
[] :: * -> *
```

```
ghci> :k [Char]
```

```
[Char] :: *
```

```
ghci> :k ()
```

```
() :: *
```

```
ghci> :k (,)
```

```
(,) :: * -> * -> *
```

```
ghci> :k (,,)
```

```
(,,) :: * -> * -> * -> *
```

Класс типов Foldable

```
class Foldable ( t :: * -> * ) where  
  fold :: Monoid m => t m -> m  
  foldMap :: Monoid m => (a -> m) -> t a -> m  
  foldr :: (a -> b -> b) -> b -> t a -> b  
  foldl :: (b -> a -> b) -> b -> t a -> b  
  foldr1 :: (a -> a -> a) -> t a -> a  
  foldl1 :: (a -> a -> a) -> t a -> a  
  toList :: t a -> [a]  
  null :: t a -> Bool  
  length :: t a -> Int  
  elem :: Eq a => a -> t a -> Bool  
  maximum :: Ord a => t a -> a  
  minimum :: Ord a => t a -> a  
  sum :: Num a => t a -> a  
  product :: Num a => t a -> a
```

Использование экземпляров класса типов Foldable

```
diameter :: (Num a, Ord a, Foldable t) => t a -> a  
diameter c = maximum c - minimum c
```

```
ghci> diameter [2, 10, 5, 44]  
42
```

```
ghci> seq = Data.Sequence.fromList [2, 10, 5, 44]  
ghci> diameter seq  
42
```

```
ghci> set = Data.Set.fromList [2, 10, 5, 44]  
ghci> diameter set  
42
```

```
ghci> diameter (Just 10)  
0
```

Свёртка контейнера с моноидом

```
ghci> :m + Data.Foldable Data.Monoid
ghci> let xs = [1,2,3,4,5]
ghci> fold (map Sum xs)
Sum {getSum = 15}
ghci> fold (map Product xs)
Product {getProduct = 120}
```

Второй пример абстрактного процесса обработки

```
process2 :: (Eq m, Monoid m, Foldable t) =>  
          m -> m -> t m -> m
```

```
process2 x y zs  
  | x<>y == mempty = mempty  
  | otherwise = fold zs
```

Foldable для кортежей



Johan Tibell

@johtib



Читаю

Prelude> length (1,2)

1

[#haskell-wat](#)

Показать перевод

РЕТВИТОВ

14

ИЗБРАННОЕ

22



0:31 - 14 окт. 2015 г.



Foldable для кортежей: объяснение

```
ghci> :k (,)
(,) :: * -> * -> *
ghci> :k (,) Int
(,) Int :: * -> *
ghci> :k (,) Int Int
(,) Int Int :: *
```

```
class Foldable ( t :: * -> * ) where
  ...
  length :: t a -> Int
  ...
```

- Eq, Ord, Show
- Semigroup, Monoid
- Foldable
- Functor, Applicative, Monad