Функциональное программирование
Лекция 6. Классы типов

Денис Николаевич Москвин

ИТМО, магистратура JB SE

12.10.2020

План лекции

1. Виды полиморфизма
2. Классы типов
3. Стандартные классы типов
4. Внутренняя реализация классов типов

План лекции

1. Виды полиморфизма
2. Классы типов
3. Стандартные классы типов
4. Внутренняя реализация классов типов

Параметрический полиморфизм

Рассмотрим функцию

```
id    :: a -> a
id x  =  x
```

Её код универсален, то есть

- годен для использования с параметром любого типа;
- не зависит ни от какой специфики этого типа.

```
id True      :: Bool
id "Hello"   :: [Char]
id id        :: a -> a
```

Ограничения полиморфизма

Рассмотрим функцию, определяющую, имеется ли элемент в списке:

```
elem           :: a -> [a] -> Bool
elem _ []      =  False
elem x (y:ys)  =  x == y || elem x ys
```

Для любого ли типа элементов она подходит?

Ограничения полиморфизма

Рассмотрим функцию, определяющую, имеется ли элемент в списке:

```
elem           :: a -> [a] -> Bool
elem _ []      =  False
elem x (y:ys)  =  x == y || elem x ys
```

Для любого ли типа элементов она подходит?

Да, но только если оператор равенства универсален:

```
(==) :: a -> a -> Bool
```

Хорошо ли это?

Сравнимость выражений

Рассмотрим, например, две разные реализации функции следования

```
suc :: (forall a.(a -> a) -> a -> a) -> (a -> a) -> a -> a
suc  = \n s z -> n s (s z)
```

```
suc' :: (forall a.(a -> a) -> a -> a) -> (a -> a) -> a -> a
suc' = \n s z -> s (n s z)
```

Можно ли утверждать, что

```
suc ≡ suc'
```

Сравнимость выражений

Рассмотрим, например, две разные реализации функции следования

```
suc :: (forall a.(a -> a) -> a -> a) -> (a -> a) -> a -> a
suc  = \n s z -> n s (s z)
```

```
suc' :: (forall a.(a -> a) -> a -> a) -> (a -> a) -> a -> a
suc' = \n s z -> s (n s z)
```

Можно ли утверждать, что

```
suc ≡ suc'
```

Эти функции равны *экстенсионально*, но не *интенсионально*.

Экстенсиональное равенство

Результат вычисления на любых входах — один и тот же:

```
zero, one, two :: (a -> a) -> a -> a
zero = \s z -> z
one  = \s z -> s z
two  = \s z -> s (s z)
```

```
GHCi> suc two (+1) 0
3
GHCi> suc' two (+1) 0
3
GHCi> suc two ('A':) ""
"AAA"
GHCi> suc' two ('A':) ""
"AAA"
```

Специальный (ad hoc) полиморфизм

- Специальный (ad hoc) полиморфизм — вид полиморфизма, противоположный параметрическому (Кристофер Стрейчи, 1967).
- Интерфейс общий (полиморфный), но реализация специализирована для каждого конкретного типа:

```
GHCi> 3 :: Integer
3
GHCi> 3 :: Double
3.0
GHCi> 3 :: Rational
3 % 1
GHCi> 3 :: Char
<interactive>: error:
  * No instance for (Num Char) arising from the literal `3'
```

План лекции

1. Виды полиморфизма
2. Классы типов
3. Стандартные классы типов
4. Внутренняя реализация классов типов

Классы типов

Класс типов — это именованный набор имён функций с сигнатурами, параметризованными общим тѝповым параметром:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Имя класса типов задаёт ограничение, называемое *контекстом*:

```
(==)            :: Eq a => a -> a -> Bool

elem            :: Eq a => a -> [a] -> Bool
elem _ []       =  False
elem x (y:ys)   =  x == y || elem x ys
```

Объявления представителей (instance declarations)

Тип a является *представителем* класса, если для него реализованы определения функций этого класса:

```
instance Eq Bool where
  True  == True   = True
  False == False  = True
  _     == _      = False
  x     /= y      = not (x == y)
```

```
instance Eq Char where
  (C# c1) == (C# c2) = c1 `eqChar#` c2
  (C# c1) /= (C# c2) = c1 `neChar#` c2
```

Символ # указывает на то, что тип данных unboxed (удерживаются не через указатель) и, следовательно, unlifted (не может быть $\perp$).

Полиморфизм при объявлении представителей

- Тип-представитель класса может быть полиморфным

```
instance Eq a => Eq [a] where
  []     == []     =  True
  (x:xs) == (y:ys) =  x == y && xs == ys
  _      == _      =  False
```

- Контекст (в данном случае `Eq a =>`) можно использовать при объявлении представителя.
- Без указания контекста такое определение приведёт к ошибке при проверке типов.

Методы по умолчанию

- Выше мы могли определять перегрузку только (==), поскольку в определении класса типов Eq имеется реализация по умолчанию для метода (/=)

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

- Методы по умолчанию могут быть перегружены в объявлениях представителя (например, из соображений эффективности).

Производные представители (derived instances)

```
data Point a = Point a a deriving Eq
```

```
GHCi> Point 3 5 == Point 3 2
False
GHCi> Point 3 5 == Point 3.0 5.0
True
GHCi> Point 3 5 == Point 'a' 'b'
<interactive>:1:9:
    No instance for (Num Char) ...
```

Задав ключ `-XStandaloneDeriving` в прагме `OPTIONS_GHC` можно использовать отдельностоящие объявления

```
deriving instance Show a => Show (Point a)
```

Расширение класса (class extension)

- Класс `Ord` наследует все методы класса `Eq` плюс содержит собственные методы

```
class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

```
sort :: Ord a => [a] -> [a]
```

- Допустимо и множественное «наследование»

```
class (Eq a, Show a) => MyClass a where
    ...
```

Типовые операторы в объявлениях класса

Переменная типа, параметризующая класс, может иметь кайнд отличный от *

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
instance Functor [] where
  fmap = map
```

```
instance  Functor Maybe  where
  fmap _ Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

Представители-сироты (orphan instances)

- Представителя класса типов `C` для типа данных `T` принято объявлять либо в модуле, где определен `C`, либо в модуле с определением `T`.
- Определение в других модулях допустимо, но потенциально небезопасно. Таких представителей называют сиротами (orphan instances).
- Проблема в том, что представители — неименованные сущности, поэтому явно управлять их доступностью через списки экспорта и импорта невозможно.
- GHC при компиляции выдает предупреждение об обнаруженной сиротливости.

Сравнение с другими языками

- В ООП-языках классы содержат и данные и методы; в Haskell'е их определения разнесены.
- Методы класов в Haskell'е напоминают виртуальные функции в C++.
- Классы типов похожи на интерфейсы в Java. Они определяют протокол использывания объекта, а не сам объект.

План лекции

1. Виды полиморфизма
2. Классы типов
3. Стандартные классы типов
4. Внутренняя реализация классов типов

Класс Ord

Минимальное полное определение: compare или <=.

```
class  (Eq a) => Ord a  where
  compare              :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a

  compare x y = if x == y then EQ
                else if x <= y then LT
                else GT
  x <  y = case compare x y of {LT -> True; _ -> False}
  x <= y = case compare x y of {GT -> False; _ -> True}
  x >  y = case compare x y of {GT -> True; _ -> False}
  x >= y = case compare x y of {LT -> False; _ -> True}
  max x y = if x <= y then y else x
  min x y = if x <= y then x else y
```

Классы Enum и Bounded

Минимальное полное определение: `toEnum` и `fromEnum`.

```
class  Enum a   where
  succ, pred      :: a -> a
  toEnum          :: Int -> a
  fromEnum        :: a -> Int

  enumFrom        :: a -> [a]             -- [n..]
  enumFromThen    :: a -> a -> [a]        -- [n,n'..]
  enumFromTo      :: a -> a -> [a]        -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a]    -- [n,n'..m]
```

```
class  Bounded a  where
  minBound, maxBound :: a
```

Класс Num

Минимальное полное определение: все, кроме `negate` или `(-)`.

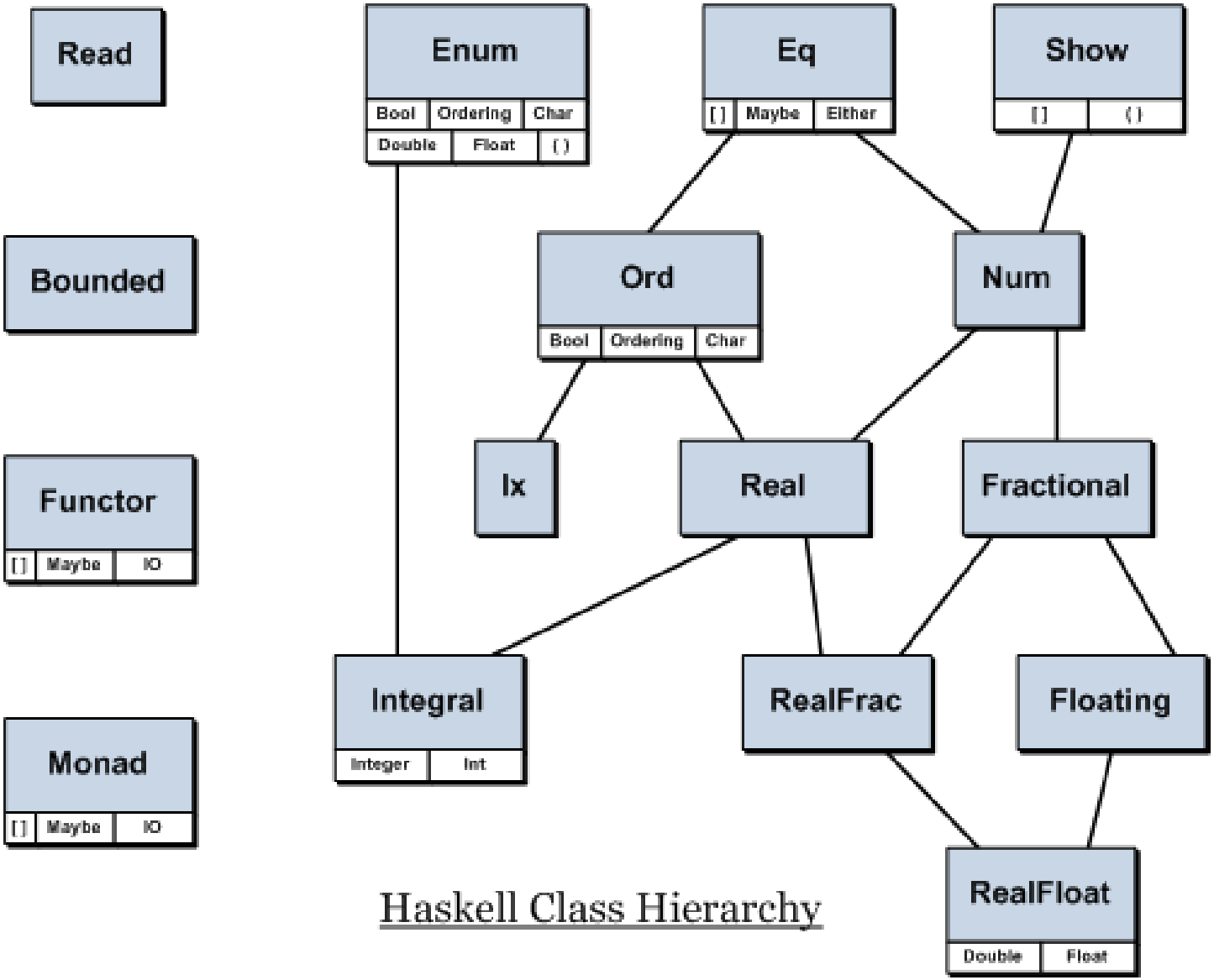
```
class (Eq a, Show a) => Num a where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a

    x - y = x + negate y
    negate x = 0 - x
```

Контекста `Ord` нет — для комплексных, например, он лишний. В GHCi уже давно нет даже контекста `(Eq a, Show a)`. Есть надежда, что следующий стандарт отразит это разумное решение.

Подклассы класса Num

- У `Num` два главных подкласса:
  - `Integral` — целочисленное деление (через `Real`);
  - `Fractional` — обычное деление.
- `Integer` и `Int` — представители класса `Integral`.
- `Float` и `Double` — наследники `Fractional` через довольно длинную иерархию со множественным наследованием.
- Автоматического приведения чисел от одного типа к другому в Haskell'е нет.

Стандартная иерархия классов типов



Haskell Class Hierarchy

Преобразования от целых и к целым

```
GHCi> :t fromIntegral
fromIntegral :: (Num b, Integral a) => a -> b
GHCi> :t sqrt
sqrt :: Floating a => a -> a
GHCi> sqrt 4
2.0
GHCi> sqrt (4::Int)
<interactive>:1:1:
    No instance for (Floating Int) ...
GHCi> sqrt $ fromIntegral (4::Int)
2.0
```

В обратную сторону (класс `RealFrac`)

```
ceiling, floor, truncate, round
    :: (RealFrac a, Integral b) => a -> b
```

Преобразования к рациональным дробям

```
data  Ratio a = !a :% !a  deriving (Eq)
(%)                       :: (Integral a) => a -> a -> Ratio a
numerator, denominator :: (Integral a) => Ratio a -> a

type  Rational  =  Ratio Integer
```

```
GHCi> :t toRational
toRational :: Real a => a -> Rational
GHCi> toRational 2.5
5 % 2
GHCi> 10 % 5
<interactive>:1:4: Not in scope: `%'
GHCi> :m +Data.Ratio
GHCi> 1 % 3 + 1 % 6
1 % 2
```

Преобразования к рациональным дробям

Числа с плавающей точкой лучше, конечно, не преобразовывать, а аппроксимировать:

```
GHCi> toRational 4.9
2758547717644 29 % 562949953421312
GHCi> approxRational 4.9 0.1
5 % 1
GHCi> approxRational 4.9 0.01
49 % 10
```

Механизм `default` трактует `4.9` как `Double` при вычислении `toRational 4.9`.
Но сами по себе числовые литералы с плавающей хранятся как вызов `fromRational (lit :: Rational)`. Поэтому

```
GHCi> 4.9 :: Rational
49 % 10
```

План лекции

1. Виды полиморфизма
2. Классы типов
3. Стандартные классы типов
4. Внутренняя реализация классов типов

Реализация классов типов: словари

- Классы типов реализуются через механизм передачи словарей (Dictionaries).
- Словарь для класса — это запись из его методов

```
data Eq' a = MkEq { eq, ne :: a -> a -> Bool }
```

- Функции-селекторы выбирают методы равенства и неравенства из этого словаря

```
GHCi> :t eq
eq :: Eq' a -> a -> a -> Bool
GHCi> :t ne
ne :: Eq' a -> a -> a -> Bool
```

Реализация объявлений представителей

- Объявления представителей транслируются в функции, возвращающие словарь...

```
dEqInt :: Eq' Int
dEqInt =  MkEq {
    eq = eqInt,
    ne = \x y -> not $ eqInt x y
    }
```

- ... или в функции, принимающие некоторый словарь и возвращающие более сложный словарь

```
dEqList              :: Eq' a -> Eq' [a]
dEqList (MkEq e _) =  MkEq el (\x y -> not $ el x y)
    where el []     []     = True
          el (x:xs) (y:ys) = x `e` y && xs `el` ys
          el _      _      = False
```

Использование словаря вместо контекста

elem теперь принимает словарь в качестве явного параметра

```
elem'               :: Eq' a -> a -> [a] -> Bool
elem' _ _ []        =  False
elem' d x (y:ys)    =  eq d x y || elem' d x ys
```

```
GHCi> elem' dEqInt 2 [3,5,2]
True
GHCi> elem' dEqInt 2 [3,5,7]
False
GHCi> elem' (dEqList dEqInt) [3,5] [[4],[1,2,3],[3,5]]
True
GHCi> elem' (dEqList dEqInt) [3,5] [[4],[1,2,3],[3,8]]
False
```