Функциональное программирование
Лекция 6. Классы типов

Денис Николаевич Москвин

СПбГУ, факультет МКН,
бакалавриат «Современное программирование», 2 курс

16.10.2025

План лекции

План лекции

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> :t 3
3 :: Num p => p
GHCi> 3 :: Integer
3
GHCi> 3 :: Double
3.0
GHCi> 3 :: Rational
3 % 1
GHCi> 3 :: Char
error: No instance for (Num Char) arising from the literal `3'
```

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 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.

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.
- Хотя реализации для (`\=`) в представителе нет, этот код скомпилируется! Почему? См. след. слайд.

Методы класса по умолчанию

```haskell
class  Eq a  where
  (==), (/=) :: a -> a -> Bool
  x /= y  = not (x == y)
  x == y  = not (x /= y)
  {-# MINIMAL (==) | (/=) #-}
```

Методы по умолчанию могут быть перегружены в объявлениях представителя.

```
GHCi> data UU = UU
GHCi> instance Eq UU where
<interactive>:4:10: warning: [-Wmissing-methods]
    * No explicit implementation for either `==' or `/='
    * In the instance declaration for `Eq UU'
GHCi> UU /= UU
Interrupted.
```

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

Законы для классов типов

Стандарт говорит, что класс типов `Ord` задает линейный порядок на типах (totally ordered).
Такие требования к классу типов оформляют в виде законов

```
∀x. x <= x                                  -- Reflexivity

∀x y z. x <= y && y <= z  ≡  x <= z         -- Transitivity

∀x y. x <= y && y <= x  ≡  x == y           -- Antisymmetry

∀x y. x <= y || y <= x                      -- Comparability
```

Подразумевается, что законы верны для любого представителя класса типов; оптимизатор может их использовать.
Однако проверка «законности» пользовательского представителя остается на совести программиста.

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.

Производные представители для newtype

Расширение GeneralizedNewtypeDeriving позволяет автоматически генерировать представителя класса типов Num.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype Temperature = Temperature {getTemp :: Double}
  deriving (Num,Eq,Show)
```

```
GHCi> Temperature 23 - Temperature 3
Temperature {getTemp = 20.0}
```

```
{-# LANGUAGE DerivingStrategies #-}
newtype Temperature = Temperature {getTemp :: Double}
 deriving (Num,Eq)
 deriving newtype Show
```

```
GHCi> Temperature 23 - Temperature 3
20.0
```

Фантомные типы

Фантомные типы позволяют хранить в типе дополнительную информацию, используемую при проверке типов.

```
newtype Temperature a = Temperature {getTemp :: Double}
 deriving (Num,Eq)
 deriving newtype Show

data Celsius
data Fahrenheit

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

План лекции

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 `(-)`.

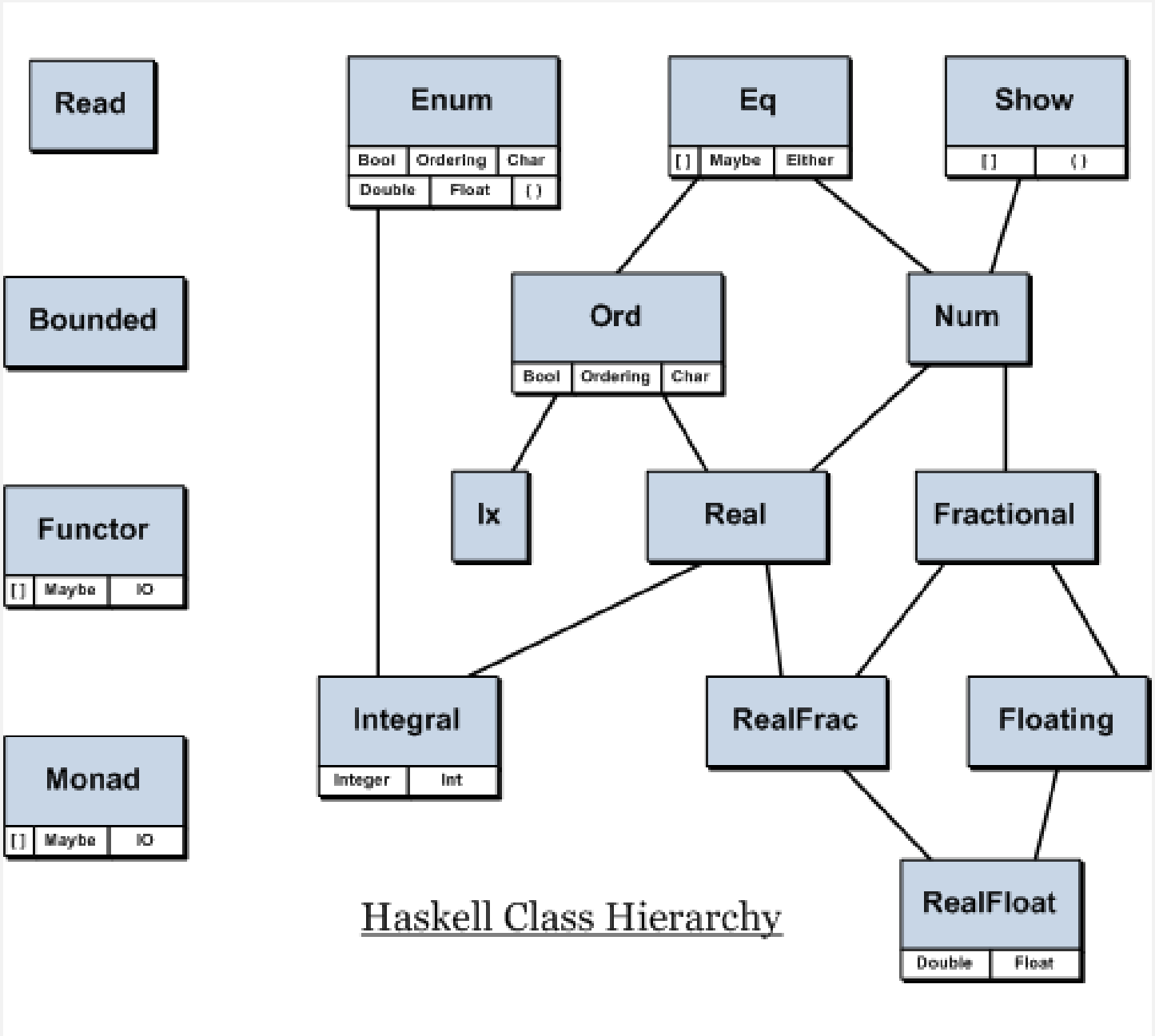
```
class (Eq a, Show a) => Num a where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a

    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.
- Типы данных `Integer` и `Int` — представители класса `Integral`.
- Типы данных `Float` и `Double` — представители класса `Fractional`.
- Автоматического приведения чисел от одного типа к другому в Haskell'е нет.

Стандартная иерархия классов типов



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

```
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> import 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
2758454771764429 % 562949953421312
GHCi> approxRational 4.9 0.1
5 % 1
GHCi> approxRational 4.9 0.01
49 % 10
```

План лекции

Реализация классов типов: словари

Рассмотрим класс типов `EQ`, изоморфный стандартному `Eq`

```
class EQ a where
  eq :: a -> a -> Bool
  ne :: a -> a -> Bool
```

Реализация классов типов: словари

Рассмотрим класс типов EQ, изоморфный стандартному Eq

```
class EQ a where
  eq :: a -> a -> Bool
  ne :: a -> a -> Bool
```

Он транслируется в словарь (dictionary), представляющий собой запись из методов класса

```
data EQ a = MkEQ {
  eq :: a -> a -> Bool,
  ne :: a -> a -> Bool }
```

Метки полей выбирают поля 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

```
instance EQ Bool where
  eq x y = case (x,y) of (True ,True)  -> True
                         (False,False) -> True
                         (_,_)         -> False
  ne x y = not $ eq x y
```

Реализация объявлений представителей

Объявления представителей

```
instance EQ Bool where
  eq x y = case (x,y) of (True ,True)  -> True
                         (False,False) -> True
                         (_,_)         -> False
  ne x y = not $ eq x y
```

транслируются в конкретные словари

```
dEQBool :: EQ Bool
dEQBool =  MkEQ {
  eq = \x y -> case (x,y) of (True ,True)  -> True
                             (False,False) -> True
                             (_,_)         -> False,
  ne = \x y -> not $ eq dEQBool x y }
```

Представители с контекстами

Объявления представителей с контекстами

```
instance EQ a => EQ [a] where
  eq as bs = case (as,bs) of
    ([] ,[])     -> True
    (x:xs,y:ys) -> eq x y && eq xs ys
    (_,_)        -> False
  ne x y = not $ eq x y
```

Представители с контекстами

Объявления представителей с контекстами

```
instance EQ a => EQ [a] where
  eq as bs = case (as,bs) of
    ([] ,[])     -> True
    (x:xs,y:ys) -> eq x y && eq xs ys
    (_,_)        -> False
  ne x y = not $ eq x y
```

транслируются в функции над словарями

```
dEQList :: EQ a -> EQ [a]
dEQList d = MkEQ {
    eq = \as bs -> case (as,bs) of
      ([] ,[])     -> True
      (x:xs,y:ys) -> eq d x y && eq (dEQList d) xs ys
      (_,_)        -> False,
    ne = \x y -> not $ eq (dEQList d) x y }
```

Использование словаря вместо контекста

Функции, использующие контекст,

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

Использование словаря вместо контекста

Функции, использующие контекст,

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

теперь принимают словарь в качестве явного параметра

```
ele :: EQ a -> a -> [a] -> Bool
ele _ _ [] = False
ele d x (y:ys) = eq d x y || ele d x ys
```

```
GHCi> ele dEQBool True [False,True,False]
True
GHCi> ele (dEQList dEQBool) [True,False] [[False,True,
False],[True]]
False
```