Классы типов в Haskell

Михаил Беляев

2 октября 2018 г.

Где мы сейчас

Что мы уже рассмотрели?

- **Простые**, **понятные**, **очевидные** вещи
- Синтаксис, комбинаторы, паттерн матчинг, вот это вот всё

Переходим к хардкорной части

Где мы сейчас

Что мы уже рассмотрели?

- **Простые**, **понятные**, **очевидные** вещи
- Синтаксис, комбинаторы, паттерн матчинг, вот это вот всё

Переходим к хардкорной части

Интерлюдия: полиморфизм

Что такое полиморфизм?

Интерлюдия: полиморфизм

Что такое полиморфизм?

Возможность одного и того же кода работать с разными типами данных

Интерлюдия: полиморфизм

Какой бывает полиморфизм?

- Параметрический
  Один код, любые данные
- Ad-hoc
  Разные участки кода для разных данных, но единый способ вызова

Ещё бывает динамический, но это уже совсем другая история...

Интерлюдия: полиморфизм

Какой бывает полиморфизм?

- Параметрический
  Один код, любые данные
  Generics/templates/universal types
- Ad-hoc
  Разные участки кода для разных данных, но единый способ вызова
  Overloading/type classes

Прерывание

Что такое overloading?

Классы типов

Классы типов — характерный для Haskell механизм *ad-hoc полиморфизма*

Первоисточник:
Phil Wadler,
«How to make ad-hoc polymorphism less ad-hoc»

- Чем это отличается от перегрузок?
- В чём фишка?

Классы типов: определение

Определение класса: `class`

- Набор операций, входящих в класс — это обычные функции, оперирующие значениями типа, который входит в класс
- Операции могут быть только объявлены (через сигнатуру типа) или реализованы
- В реализациях операций внутри класса можно использовать только функции, работающие с произвольными типами или объявленные (не обязательно определённые) в этом же классе

Классы типов: экземпляры

Экземпляр класса: `instance`

- Набор операций класса для *конкретного* (или обобщённого) типа
- Все операции, которые не были реализованы в классе, должны быть реализованы в экземпляре
- Операции, которые уже реализованы в классе, можно переписать или не переписывать
  (тогда будет использоваться функция из класса)

Классы типов: пример

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

data Coords = Coords Int Int
instance Eq Coords where
    (==) (Coords ax ay) (Coords bx by) =
        ax == bx && ay == by
```

Заметка

- Чем это отличается от перегрузок?

В перегрузках, как правило, можно делать что угодно.
Здесь же, как минимум, тип ограничен.

Классы типов: контроль адекватности

- Операции в классе можно переопределить как угодно, до тех пор, пока типы совпадают
- У каждого класса есть какие-то неявные законы того, как он должен «работать»:

```
(x == y && x /= y) === False для любых x и y
```

- Как правило, эти законы описаны в документации

Классы типов: использование

```
listContains (h:t) e | h == e = True
                     | otherwise = listContains t e
```

Каким должен быть тип этой функции?

Классы типов: использование

```
listContains (h:t) e | h == e = True
                     | otherwise = listContains t e
```

Каким должен быть тип этой функции?

```
listContains :: (Eq a) => [a] -> a -> Bool
```

Заметка

- Чем это отличается от перегрузок?

В перегрузках так вообще обычно нельзя (можно в C++)

Классы типов: использование

Требование наличия экземпляра класса можно предъявлять и внутри классов:

```
instance (Eq a) => Eq [a] where
    (ah:at) == (bh:bt) = ah == bh && at == bt
instance (Eq a, Eq b) => Eq (a,b) where
    (a1,a2) == (b1,b2) = a1 == b1 && a2 == b2
```

Классы типов: стандартные классы

Базовые классы типов:

- `Eq` — сравнение на равенство
- `Ord` — сравнения порядка
- `Show` — перевод в строку
- `Read` — чтение из строки
- `Bounded` — наличие минимального/максимального значения

Стандартные классы: `Eq`

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

Минимальное определение: `==` или `/=`

Закон: `a == b` $\Leftrightarrow$ `not (a /= b)`

Стандартные классы: `Ord`

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

data Ordering = LT | EQ | GT
```

Минимальное определение: `compare` или `<=`

Законы: очевидны

Стандартные классы: Show

```
type ShowS = String -> String
class Show a where
    showsPrec :: Int -> a -> ShowS
    show :: a -> String
    showList :: [a] -> ShowS
```

Минимальное определение: `show` или `showsPrec`

Законы: нет

(но неплохо бы, чтобы `show` и `read` были совместимы)

Стандартные классы: Read

```haskell
type ReadS a = String -> [(a, String)]
class Read a where
    readsPrec :: Int -> ReadS a
    readList :: ReadS [a]
    readPrec :: ReadPrec a
    readListPrec :: ReadPrec [a]
```

Минимальное определение: `readsPrec` или `readPrec`
Законы: нет
(но неплохо бы, чтобы `show` и `read` были совместимы)

Прерывание

Зачем нужны отдельно `showList` и `readList`?

Прерывание

Зачем нужны отдельно `showList` и `readList`?

Потому, что строки — это списки символов

Классы типов: классы чисел

- `Num` — числа, арифметические операции
- `Real` — действительные числа
- `Integral` — целые числа

Стандартные классы: Num

Обобщённое понятие «число»

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

Минимальное определение: `(+), (*), abs, signum, fromInteger, (negate | (-))`

Законы: соответствуют арифметическим

Прерывание

```
ghci> :type 2
(Num a) => a
```

Стандартные классы: `Real`

Рациональное число

```
data Ratio a = <compiler-dependent>
type Rational = Ratio Integer
class (Num a, Ord a) => Real a where
    toRational :: a -> Rational
```

Минимальное определение: `toRational`

Законы: нет

Стандартные классы: `Enum`

Ограниченная сущность (для оператора `(..)`)

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

Минимальное определение: `toEnum, fromEnum`

Законы: очевидны

Стандартные классы: `Integral`

Целое число

```
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
```

Минимальное определение: `quotRem, toInteger`
Законы: арифметические

Стандартные классы: `Bounded`

Любая штука, у которой есть максимум и минимум

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

Минимальное определение: `minBound, maxBound`
Законы: нет

deriving

Ключевое слово `deriving` позволяет автоматически реализовывать некоторые классы типов:
`Eq, Ord, Show, Read, Bounded, Enum`

```
data Coords = Coords{x :: Int, y :: Int}
    deriving (Eq,Ord,Show,Read)
```

-XStandaloneDeriving

```haskell
data Coords = Coords{x :: Int, y :: Int}
    deriving (Eq,Ord,Read)

deriving instance Show Coords
```

Моноид

Моноид – это тип, отвечающий двум требованиям:

- Есть нейтральное значение, называемое *нулём* моноида
- Есть операция добавления моноида к моноиду

Моноид: сигнатура

```haskell
class Semigroup a where
    (<>) :: a -> a -> a
class (Semigroup a) => Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a
```

- Основной закон моноида:

```haskell
x `mappend` mzero === mzero `mappend` x === x
```

Примеры моноидов из математики

- Целые числа, 0 и операция сложения
- Целые числа, 1 и операция умножения
- Булевы значения, False и логическое «или»
- Булевы значения, True и логическое «и»
- И т. д.

Моноиды в Haskell

- Любой контейнер, который может быть пустым
    - В частности, список, операция `append` и пустой список в качестве нуля
- `Maybe a` если `a` — моноид
- Целые и булевы значения моноидами не являются, потому что нет одного очевидного представления

Классы типов: типы высших порядков

- Haskell может оперировать не только конкретными типами и типовыми переменными

Классы типов: типы высших порядков

- Haskell может оперировать не только конкретными типами и типовыми переменными

- Можно оперировать *типами высших порядков* (higher-kind types)
  - List (вместо List a)
  - Maybe (вместо Maybe a)
  - BinaryTree (вместо BinaryTree a)
  - и т.д.

Классы типов: функтор

Вспомним операцию `map`:

```
map :: (a -> b) -> [a] -> [b]
```

Можно сказать, что `map` *отображает* функцию над обычными значениями в функцию над списками. Тип данных (в данном случае, список), поддерживающий такое отображение, называется *функтором*.

Классы типов: функтор

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

Обратите внимание: `f` здесь — не обычный тип, а обобщённый (тип с параметром).
Функторами являются все контейнеры, а также `Maybe`.

Закон:

```
fmap id x === x
```

Классы типов: функтор

```haskell
data BinaryTree a =
      EmptyBinaryTree
    | Leaf a
    | Node a (BinaryTree a) (BinaryTree a)
instance Functor BinaryTree where
    fmap f (EmptyBinaryTree) = EmptyBinaryTree
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node e l r) =
        Node (f e) (fmap f l) (fmap f r)
```

Классы типов: функтор

В качестве синонима для `fmap` определён оператор `<$>`

```
(*3) $ 2                  -- 6
(*3) <$> [1,2,3,4,5]      -- [3,6,9,12,15]
(*3) <$> (Just 5)         -- Just 15
```

Классы типов: функтор

Что, если мы хотим применить <$> к двум аргументам?

```
(*) $ 2 $ 3                    -- 6
(*) <$> [1,2,3] <$> [3,4]      -- type error!
(*) <$> Just 2 <$> Nothing     -- type error!
```

Что делать?

Классы типов: аппликативный функтор

```haskell
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

(*) $ 2 $ 3                    -- 6
(*) <$> [1,2,3] <*> [3,4]      -- ???
(*) <$> Just 2 <*> Nothing     -- ???
```

Классы типов: аппликативный функтор

```haskell
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

(*) $ 2 $ 3                    -- 6
(*) <$> [1,2,3] <*> [3,4]      -- [3,4,6,8,9,12]
(*) <$> Just 2 <*> Nothing     -- Nothing
(*) <$> Just 2 <*> Just 4      -- Just 8
```

Закон: pure f <*> x === f <$> x

Классы типов: монада

Монада — это параметризованный тип `m`, удовлетворяющий требованиям:

- Он является функтором (см. выше)
- Можно «завернуть» значение в монаду:
  `return :: a -> m a`
- Можно «расплющить» монаду от монады:
  `flatten :: m (m a) -> m a`
- Операция `bind` или `(>>=)`:
  `x >>= f = flatten (fmap f x)`
- Работают законы:

```
return x >>= f  === f x
m >>= return    === m
(m >>= f) >>= g === m >>= (\x -> f x >>= g)
```

Монада: формальное определение

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a
```

Монады: примеры

Монадой являются большинство контейнеров, Maybe и много других типов

Монада используется как абстракция *цепочки вычислений*

Но об этом — в следующей лекции

Классы типов: альтернативный функтор

Аппликативный функтор + моноид

```haskell
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Классы типов: MonadPlus

Монада + моноид

```haskell
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Summary

- В Haskell нет динамического полиморфизма
  - Как и в любом языке с **сильной** системой типов
- В Haskell нету перегрузок, но вместо них есть классы типов
  - Принципиальная разница в том, что классы типов вписаны в общую картину лучше
  - Они эффективнее взаимодействуют с другими видами полиморфизма
  - Они мощнее за счёт поддержки типов высших порядков (см. Functor)

Summary, episode 2

- Пожалуй, это самый мощный механизм статического полиморфизма, известный автору
- Почему же его не вставили во все языки?
    - Внутри много матана
        - При сложных зависимостях и сложных типах на входе
        - Многие тайпклассы приехали напрямую из теорката
        - Benjamin Pierce, Category Theory for Computer Scientists
    - Сложно для понимания среднего программиста из Мумбаи

http://kspt.icc.spbstu.ru/course/lang

belyaev@kspt.icc.spbstu.ru

КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ СПбПУ

ПОЛИТЕХ
Санкт-Петербургский политехнический университет Петра Великого