Структуры данных на языке Haskell

Григорий Волков

2022

ИСП РАН

Базовые структуры данных

Освежим память:

```
Prelude> :t (False, True, False)
(False, True, False) :: (Bool, Bool, Bool)
Prelude> :t [(1, 2), (3, 4), (5, 6)]
[(1, 2), (3, 4), (5, 6)] :: (Num a, Num b) => [(a, b)]
Prelude> :t ['h', 'i']
['h', 'i'] :: [Char]
```

Базовые структуры данных

Освежим память:

```
Prelude> :t (False, True, False)
(False, True, False) :: (Bool, Bool, Bool)
Prelude> :t [(1, 2), (3, 4), (5, 6)]
[(1, 2), (3, 4), (5, 6)] :: (Num a, Num b) => [(a, b)]
Prelude> :t ['h', 'i']
['h', 'i'] :: [Char]
```

Кстати, строки — просто списки символов:

```
Prelude> :t "hello world"
"hello world" :: [Char]
Prelude> tail "hello world"
"ello world"
Prelude> :i String
type String = [Char]
```

Базовые структуры данных

Освежим память:

```
Prelude> :t (False, True, False)
(False, True, False) :: (Bool, Bool, Bool)
Prelude> :t [(1, 2), (3, 4), (5, 6)]
[(1, 2), (3, 4), (5, 6)] :: (Num a, Num b) => [(a, b)]
Prelude> :t ['h', 'i']
['h', 'i'] :: [Char]
```

Кстати, строки — просто списки символов:

```
Prelude> :t "hello world"
"hello world" :: [Char]
Prelude> tail "hello world"
"ello world"
Prelude> :i String
type String = [Char]
```

Да, это не эффективно. Эффективные строковые типы существуют в сторонних библиотеках.

Немного подсчёта

У типа `Bool` два значения. Сколько их у `(Bool, Bool, Bool)`?

Немного подсчёта

У типа `Bool` два значения. Сколько их у `(Bool, Bool, Bool)`?

Можно подсчитать таким магическим синтаксисом как list comprehension:

```
Prelude> length [ (a,b,c) | a <- [False, True]
                          , b <- [False, True], c <- [False, True] ]
8
```

Немного подсчёта

У типа `Bool` два значения. Сколько их у `(Bool, Bool, Bool)`?

Можно подсчитать таким магическим синтаксисом как list comprehension:

```
Prelude> length [ (a,b,c) | a <- [False, True]
                          , b <- [False, True], c <- [False, True] ]
8
```

Кортеж — *тип произведения* (product type):

- `(Bool, Bool)` всегда содержит `Bool` & `Bool`
  - в алгебре логики «И» является умножением
- количество элементов считается перемножением количеств элементов входящих типов: `(Bool, (), Bool)` $\rightarrow 2 * 1 * 2 = 4$

Немного подсчёта

У типа `Bool` два значения. Сколько их у `(Bool, Bool, Bool)`?

Можно подсчитать таким магическим синтаксисом как list comprehension:

```
Prelude> length [ (a,b,c) | a <- [False, True]
                          , b <- [False, True], c <- [False, True] ]
8
```

Кортеж — *тип произведения* (product type):

- `(Bool, Bool)` всегда содержит `Bool` & `Bool`
  - в алгебре логики «И» является умножением
- количество элементов считается перемножением количеств элементов входящих типов: `(Bool, (), Bool)` $\rightarrow 2 * 1 * 2 = 4$

С `Bool` как-то не интересно, научимся вводить перечисления (`enum` в других языках):

```
data Access = Guest | User | Moderator | Admin
```

Тогда `(Access, Bool)` $\rightarrow 4 * 2 = 8$.

Немного подсчёта

У типа `Bool` два значения. Сколько их у `(Bool, Bool, Bool)`?

Можно подсчитать таким магическим синтаксисом как list comprehension:

```
Prelude> length [ (a,b,c) | a <- [False, True]
                          , b <- [False, True], c <- [False, True] ]
8
```

Кортеж — *тип произведения* (product type):

- `(Bool, Bool)` всегда содержит `Bool` & `Bool`
  - в алгебре логики «И» является умножением
- количество элементов считается перемножением количеств элементов входящих типов: `(Bool, (), Bool)` $\rightarrow 2 * 1 * 2 = 4$

С `Bool` как-то не интересно, научимся вводить перечисления (`enum` в других языках):

```
data Access = Guest | User | Moderator | Admin
```

Тогда `(Access, Bool)` $\rightarrow 4 * 2 = 8$. `Bool` — частный случай перечисления:

```
data Bool = False | True
```

Вводим записи

Если рассмотрели `enum`, стоит рассмотреть и `struct`. В Haskell это называется записями (records):

```
data Process = Process { pid     :: Int
                       , creds   :: (Int, Int)
                       , command :: String
                       , running :: Bool }
```

Это тоже тип произведения. Это примерно то же самое, что `(Int, (Int, Int), String, Bool)`, только с названными полями.

Вводим записи

Если рассмотрели `enum`, стоит рассмотреть и `struct`. В Haskell это называется записями (records):

```
data Process = Process { pid     :: Int
                       , creds   :: (Int, Int)
                       , command :: String
                       , running :: Bool }
```

Это тоже тип произведения. Это примерно то же самое, что `(Int, (Int, Int), String, Bool)`, только с названными полями.

Название два раза? То же ключевое слово `data`, что и для перечислений??

Давайте сначала.

Конструкция data

С самого начала, с самого простого. Можно задать тип без значений:

```
data EmptyType
```

Конструкция data

С самого начала, с самого простого. Можно задать тип без значений:

```
data EmptyType
```

Можно задать тип с одним значением:

```
data UnitType = TheOneUnitValue
```

Конструкция data

С самого начала, с самого простого. Можно задать тип без значений:

```
data EmptyType
```

Можно задать тип с одним значением:

```
data UnitType = TheOneUnitValue
```

Можно перечислить допустимые значения:

```
data EnumType = One | Two | Three | Four
```

Конструкция data

С самого начала, с самого простого. Можно задать тип без значений:

```
data EmptyType
```

Можно задать тип с одним значением:

```
data UnitType = TheOneUnitValue
```

Можно перечислить допустимые значения:

```
data EnumType = One | Two | Three | Four
```

Можно задать значение с полями — подобие кортежа:

```
data PairType = IBPair Int Bool
```

```
Prelude> :t IBPair 123 True
IBPair 123 True :: PairType
```

Конструкция data

С самого начала, с самого простого. Можно задать тип без значений:

```
data EmptyType
```

Можно задать тип с одним значением:

```
data UnitType = TheOneUnitValue
```

Можно перечислить допустимые значения:

```
data EnumType = One | Two | Three | Four
```

Можно задать значение с полями — подобие кортежа:

```
data PairType = IBPair Int Bool
```

```
Prelude> :t IBPair 123 True
IBPair 123 True :: PairType
```

Можно перечислить допустимые значения... у каждого из которых могут быть поля!!

```
data PairsType = IBPair Int Bool | BBPair Bool Bool
```

Алгебраические типы данных

Синтаксис `data` в общем случае задаёт тип как *сумму произведений* типов!

```
data PairsType = IBPair Int Bool | BBPair Bool Bool
```

В данном примере значение — (`Int` И `Bool`) ИЛИ (`Bool` И `Bool`).

Алгебраические типы данных

Синтаксис `data` в общем случае задаёт тип как *сумму произведений* типов!

```haskell
data PairsType = IBPair Int Bool | BBPair Bool Bool
```

В данном примере значение — (`Int` И `Bool`) ИЛИ (`Bool` И `Bool`).
Количество — `(maxBound :: Int) * 2 + 2 * 2` :)

Алгебраические типы данных

Синтаксис `data` в общем случае задаёт тип как *сумму произведений* типов!

```
data PairsType = IBPair Int Bool | BBPair Bool Bool
```

В данном примере значение — (`Int` И `Bool`) ИЛИ (`Bool` И `Bool`).
Количество — `(maxBound :: Int) * 2 + 2 * 2` :)

А почему значения выглядят так?

```
Prelude> :t IBPair 123 True
IBPair 123 True :: PairsType
```

Алгебраические типы данных

Синтаксис `data` в общем случае задаёт тип как *сумму произведений* типов!

```
data PairsType = IBPair Int Bool | BBPair Bool Bool
```

В данном примере значение — (`Int` И `Bool`) ИЛИ (`Bool` И `Bool`).
Количество — `(maxBound :: Int) * 2 + 2 * 2` :)

А почему значения выглядят так?

```
Prelude> :t IBPair 123 True
IBPair 123 True :: PairsType
```

```
Prelude> :t IBPair 123
IBPair 123 :: Bool -> PairsType
Prelude> :t IBPair
IBPair :: Int -> Bool -> PairsType
```

Конечно, и тут функции! Эти функции называются *конструкторами данных*.

Алгебраические типы данных

Синтаксис `data` в общем случае задаёт тип как *сумму произведений* типов!

```
data PairsType = IBPair Int Bool | BBPair Bool Bool
```

В данном примере значение — (`Int` И `Bool`) ИЛИ (`Bool` И `Bool`).
Количество — `(maxBound :: Int) * 2 + 2 * 2` :)

А почему значения выглядят так?

```
Prelude> :t IBPair 123 True
IBPair 123 True :: PairsType
```

```
Prelude> :t IBPair 123
IBPair 123 :: Bool -> PairsType
Prelude> :t IBPair
IBPair :: Int -> Bool -> PairsType
```

Конечно, и тут функции! Эти функции называются *конструкторами данных*.

Но они являются не только функциями, но и паттернами:

```
getBools (IBPair _ b) = [b]
getBools (BBPair a b) = [a, b]
```

И все-таки о записях

Вернёмся к синтаксису записей:

```
data Process = Process { pid     :: Int
                       , creds   :: (Int, Int)
                       , command :: String
                       , running :: Bool }
```

И все-таки о записях

Вернёмся к синтаксису записей:

```
data Process = Process { pid     :: Int
                       , creds   :: (Int, Int)
                       , command :: String
                       , running :: Bool }
```

Это «синтаксический сахар», сразу определяющий функции доступа к полям:

```
data Process = Process Int (Int, Int) String Bool
pid (Process x _ _ _) = x
creds (Process _ x _ _) = x
commands (Process _ _ x _) = x
running (Process _ _ _ x) = x
```

Но не только...

Ещё о записях

Возможность использовать литералы записей:

```
Process { pid = 1234, creds = (0, 0)
        , command = "rm -rf /", running = True }
```

Ещё о записях

Возможность использовать литералы записей:

```
Process { pid = 1234, creds = (0, 0)
        , command = "rm -rf /", running = True }
```

Синтаксис обновления записей:

```
Prelude> let p1 = Process { pid = 1234, creds = (0, 0)
                          , command = "rm -rf /", running = True }
Prelude> p1 { pid = 4321 }
Process { pid = 4321, creds = (0,0)
        , command = "rm -rf /", running = True }
```

Изобретаем велосипед

Зададим свой тип списка. Это делается рекурсивно:

```
data IntList = Empty | Cons Int IntList
```

Изобретаем велосипед

Зададим свой тип списка. Это делается рекурсивно:

```
data IntList = Empty | Cons Int IntList
```

Как и функции, типы могут быть полиморфными:

```
data List a = Empty | Cons a (List a)
```

Аналогично с конструкторами данных: `List` сам по себе — *конструктор типов*, `List Int` — конкретный тип.

Изобретаем велосипед

Зададим свой тип списка. Это делается рекурсивно:

```
data IntList = Empty | Cons Int IntList
```

Как и функции, типы могут быть полиморфными:

```
data List a = Empty | Cons a (List a)
```

Аналогично с конструкторами данных: `List` сам по себе — *конструктор типов*, `List Int` — конкретный тип.

У нашего списка нет красивого синтаксиса, но это не меняет суть...

```
Prelude> Cons 1 $ Cons 2 $ Cons 3 $ Empty
Cons 1 (Cons 2 (Cons 3 Empty))
```

Операции на списках

На нашем списке можно реализовать стандартные функции, чтобы понять, как они устроены. Например `map`:

```
myMap _ Empty = Empty
myMap f (Cons x xs) = Cons (f x) (myMap f xs)
```

```
Prelude> myMap (* 2) $ Cons 1 $ Cons 2 $ Cons 3 $ Empty
Cons 2 (Cons 4 (Cons 6 Empty))
```

Операции на списках

На нашем списке можно реализовать стандартные функции, чтобы понять, как они устроены. Например `map`:

```
myMap _ Empty = Empty
myMap f (Cons x xs) = Cons (f x) (myMap f xs)
```

```
Prelude> myMap (* 2) $ Cons 1 $ Cons 2 $ Cons 3 $ Empty
Cons 2 (Cons 4 (Cons 6 Empty))
```

Рекурсивные структуры данных легко обрабатывать рекурсивными функциями. Ещё:

```
myDrop _ Empty = Empty
myDrop 0 xs = xs
myDrop n (Cons _ xs) = myDrop (n - 1) xs
```

```
Prelude> myDrop 2 $ Cons 'h' $ Cons 'i' $ Cons 'a' $ Cons 'l' $ Cons 'l' $
Cons 'a' (Cons 'l' (Cons 'l' Empty))
```

Стандартные списки

Как же всё-таки определены стандартные списки? Как разбирать их?

Стандартные списки

Как же всё-таки определены стандартные списки? Как разбирать их?

Как бы так:

```
data [a] = [] | a : [a]
```

Конечно, квадратные скобки — специальный синтаксис, но двоеточие... *двоеточие* тоже, но в общем можно определять конструкторы данных как операторы. (Так же, как и обычные функции.)

Стандартные списки

Как же всё-таки определены стандартные списки? Как разбирать их?

Как бы так:

```
data [a] = [] | a : [a]
```

Конечно, квадратные скобки — специальный синтаксис, но двоеточие... *двоеточие* тоже, но в общем можно определять конструкторы данных как операторы. (Так же, как и обычные функции.) Например, так можно:

```
infixr 1 :::
data L a = E | a ::: L a

Prelude> 1 ::: 2 ::: 3 ::: E
1 ::: (2 ::: (3 ::: E))
```

Стандартные списки

Итак, попробуем с самим стандартным списком:

```
Prelude> 1 : 2 : 3 : []
[1,2,3]
```

Соответственно `map`:

```
myMap _ [] = []
myMap f (x : xs) = f x : myMap f xs

Prelude> myMap (* 2) [1,2,3,4]
[2,4,6,8]
```

Упражнение: filter

```
myMap _ [] = []
myMap f (x : xs) = f x : myMap f xs
```

A `filter`?

```
myFilter _ [] = []
myFilter p (x : xs) =
```

Упражнение: filter

```
myMap _ [] = []
myMap f (x : xs) = f x : myMap f xs
```

A `filter`?

```
myFilter _ [] = []
myFilter p (x : xs) =
  if p x then x : myFilter p xs else myFilter p xs
```

```
Prelude> myFilter (\x -> x `mod` 2 == 0) [1..10]
[2,4,6,8,10]
```

Полезные стандартные типы: Maybe

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

«Значение, которое может отсутствовать» — вместо `null`. Такой тип в других языках называется `Option` или `Nullable`.

Полезные стандартные типы: Maybe

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

«Значение, которое может отсутствовать» — вместо `null`. Такой тип в других языках называется `Option` или `Nullable`.

Отличный пример использования: безопасная версия функции `head`:

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

Полезные стандартные типы: Maybe

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

«Значение, которое может отсутствовать» — вместо `null`. Такой тип в других языках называется `Option` или `Nullable`.

Отличный пример использования: безопасная версия функции `head`:

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

Интересные функции в стандартной библиотеке позволяют легко делать, например, значение по умолчанию:

```
Prelude> import Data.Maybe
Prelude Data.Maybe> fromMaybe 0 $ safeHead [1..10]
1
Prelude Data.Maybe> fromMaybe 0 $ safeHead []
0
```

Полезные стандартные типы: Either

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

«Или одно, или другое» — чаще всего используется для ошибок.

```haskell
safeDiv :: Float -> Float -> Either String Float
safeDiv x 0 = Left "division by zero"
safeDiv x y = Right (x / y)
```

При таком использовании «успешное» значение принято заворачивать в конструктор `Right`.

Класс типов Show

В предыдущих слайдах я опять немного вас обманул :)

```
Prelude> let p1 = Process { pid = 1234, creds = (0, 0)
                          , command = "rm -rf /", running = True }
Prelude> p1 { pid = 4321 }
<interactive>:16:1: error:
    • No instance for (Show Process) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

Класс типов Show

В предыдущих слайдах я опять немного вас обманул :)

```
Prelude> let p1 = Process { pid = 1234, creds = (0, 0)
                          , command = "rm -rf /", running = True }
Prelude> p1 { pid = 4321 }
<interactive>:16:1: error:
    • No instance for (Show Process) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

Чтобы работало, тип должен принадлежать классу Show, отвечающему за печать данных в строковое представление.

Это как правило делается автоматическим выводом (deriving):

```
data Process = … deriving (Show)
```

deriving (ещё что-нибудь)

Компилятор умеет автоматически выводить многие экземпляры классов типов:

```
data Nums = Ints Int Int
          | Floats Float Float
          deriving (Show, Ord, Eq)
```

Как думаете, как будет работать сравнение?

deriving (ещё что-нибудь)

Компилятор умеет автоматически выводить многие экземпляры классов типов:

```
data Nums = Ints Int Int
          | Floats Float Float
          deriving (Show, Ord, Eq)
```

Как думаете, как будет работать сравнение?

Первичны конструкторы! (их порядок в определении типа)

```
Prelude> Ints 1 1 < Floats 2 2
True
```

Дальше сравниваются поля по порядку.

```
Prelude> Ints 1 2 `compare` Ints 2 1
LT
```

Подробнее о классах типов

Наконец мы подобрались к этому вопросу! Что такое typeclass? Спросим GHCi про какой-нибудь знакомый:

```
Prelude> :i Eq
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
```

Перед нами связанный *набор сигнатур функций*, заданный относительно одного общего параметра `a`.

И какая-то магическая аннотация `MINIMAL`.

Подробнее о классах типов

Наконец мы подобрались к этому вопросу! Что такое typeclass? Спросим GHCi про какой-нибудь знакомый:

```
Prelude> :i Eq
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
```

Перед нами связанный *набор сигнатур функций*, заданный относительно одного общего параметра a.

И какая-то магическая аннотация `MINIMAL`. Она задаёт минимальный набор функций, которые надо реализовать. Это используется для предупреждений и документации. А почему вообще можно задать не все??

Реализации по умолчанию

Потому что — так тут и сделано — могут быть реализации по умолчанию:

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

Eq задаёт операторы равенства и неравенства по умолчанию в терминах друг друга, чтобы при написании экземпляра класса можно было реализовать *любой* из них.

Определение экземпляров классов типов

Попробуем с каким-нибудь простым типом:

```
data MyPair = MP Int String

instance Eq MyPair where
  MP a b == MP a' b' = a == a' && b == b'
```

Определение экземпляров классов типов

Попробуем с каким-нибудь простым типом:

```
data MyPair = MP Int String


instance Eq MyPair where
  MP a b == MP a' b' = a == a' && b == b'
```

Если инфиксный (операторный) синтаксис немного напугал, посмотрим что-нибудь более «обычное»:

```
instance Show MyPair where
  show (MP a b) = "MP " ++ show a ++ " " ++ show b
```

Неочевидные особенности

В отличие от интерфейсов в Java/C# и т.д. наш `instance`, связывающий класс типов и тип — обособленная конструкция. Где его можно определять?

Неочевидные особенности

В отличие от интерфейсов в Java/C# и т.д. наш `instance`, связывающий класс типов и тип — обособленная конструкция. Где его можно определять?

Мы пока писали рядом с типом, но можно и рядом с классом — для любых доступных типов вообще:

```haskell
class Quadruplable a where
  quadruple :: a -> a

instance Quadruplable Int where
  quadruple = (* 4)

instance Quadruplable [a] where
  quadruple xs = xs ++ xs ++ xs ++ xs
```

Неочевидные особенности

В отличие от интерфейсов в Java/C# и т.д. наш `instance`, связывающий класс типов и тип — обособленная конструкция. Где его можно определять?

Мы пока писали рядом с типом, но можно и рядом с классом — для любых доступных типов вообще:

```haskell
class Quadruplable a where
  quadruple :: a -> a

instance Quadruplable Int where
  quadruple = (* 4)

instance Quadruplable [a] where
  quadruple xs = xs ++ xs ++ xs ++ xs
```

А можно где угодно. Но осторожно. Компилятор будет предупреждать, что такие «экземпляры-сироты» (orphan instances) могут приводить к конфликтам. Но иногда для интеграции библиотек это бывает необходимо.

Совсем неочевидные особенности

А что если в предыдущем примере сделать для всех числовых типов?

```
class Quadruplable a where
  quadruple :: a -> a

instance Num a => Quadruplable a where
  quadruple = (* 4)

instance Quadruplable [a] where
  quadruple xs = xs ++ xs ++ xs ++ xs
```

Совсем неочевидные особенности

А что если в предыдущем примере сделать для всех числовых типов?

```
class Quadruplable a where
  quadruple :: a -> a

instance Num a => Quadruplable a where
  quadruple = (* 4)

instance Quadruplable [a] where
  quadruple xs = xs ++ xs ++ xs ++ xs
```

```
<interactive>:144:19: error:
    • Illegal instance declaration for ‘Quadruplable a’
        (All instance types must be of the form (T a1 ... an)
         where a1 ... an are *distinct type variables*,
         and each type variable appears at most once in the instance head.
         Use FlexibleInstances if you want to disable this.)
    • In the instance declaration for ‘Quadruplable a’
```

Расширения языка

Компилятор подсказал расширение языка. Вот так его можно включить в GHCi:

```
Prelude> :set -XFlexibleInstances
```

А так в модуле (на верху файла):

```
{-# LANGUAGE FlexibleInstances #-}
```

Что будет с ним?

Расширения языка

Компилятор подсказал расширение языка. Вот так его можно включить в GHCi:

```
Prelude> :set -XFlexibleInstances
```

А так в модуле (на верху файла):

```
{-# LANGUAGE FlexibleInstances #-}
```

Что будет с ним?

```
<interactive>:155:10: error:
    • The constraint ‘Num a’
        is no smaller than the instance head ‘Quadruplable a’
      (Use UndecidableInstances to permit this)
    • In the instance declaration for ‘Quadruplable a’
```

Тут уже совсем плохие слова написаны, но продолжим и с этим расширением.

Расширяем механизм классов типов?

```
Prelude> quadruple 4.2
16.8
Prelude> quadruple [1,2]

<interactive>:179:1: error:
    • Non type-variable argument in the constraint: Quadruplable [a]
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Quadruplable [a], Num a) => [a]
```

Расширяем механизм классов типов?

```
Prelude> quadruple 4.2
16.8
Prelude> quadruple [1,2]

<interactive>:179:1: error:
    • Non type-variable argument in the constraint: Quadruplable [a]
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Quadruplable [a], Num a) => [a]


Prelude> :set -XFlexibleContexts
Prelude> quadruple [1,2]

<interactive>:181:1: error:
    • Overlapping instances for Quadruplable [Integer]
        arising from a use of ‘it’
      Matching instances:
        instance [safe] Num a => Quadruplable a
        instance [safe] Quadruplable [a]
```

Почему?

Ну почему эти экземпляры перекрываются??

```
instance Num a => Quadruplable a
instance Quadruplable [a]
```

Почему?

Ну почему эти экземпляры перекрываются??

```
instance Num a => Quadruplable a
instance Quadruplable [a]
```

Догадаться нетрудно: значит «левая часть» (ограничения) **не учитываются** при поиске экземпляра, а всего лишь проверяются потом!

На то они и ограничения... можно ли с этим справиться?

Почему?

Ну почему эти экземпляры перекрываются??

```
instance Num a => Quadruplable a
instance Quadruplable [a]
```

Догадаться нетрудно: значит «левая часть» (ограничения) **не учитываются** при поиске экземпляра, а всего лишь проверяются потом!

На то они и ограничения... можно ли с этим справиться? Да:

```
class Quadruplable a where
  quadruple :: a -> a

instance {-# OVERLAPPABLE #-} Num a => Quadruplable a where
  quadruple = (* 4)

instance {-# OVERLAPS #-} Quadruplable [a] where
  quadruple xs = xs ++ xs ++ xs ++ xs
```

Надо ли так делать? Нет, конечно такие ситуации надо избегать. ~~Но опять же иногда приходится...~~

Больше интересных стандартных классов типов

Говорят, в Haskell много абстракций из математики... а вот например полугруппа:

```haskell
class Semigroup a where
  -- | An associative operation.
  (<>) :: a -> a -> a
```

Всё, что требуется — ассоциативная операция. Как правило она «сочетает» значения. Классический пример:

```haskell
instance Semigroup [a] where (<>) = (++)
```

Больше интересных стандартных классов типов

Говорят, в Haskell много абстракций из математики... а вот например полугруппа:

```haskell
class Semigroup a where
  -- | An associative operation.
  (<>) :: a -> a -> a
```

Всё, что требуется — ассоциативная операция. Как правило она «сочетает» значения. Классический пример:

```haskell
instance Semigroup [a] where (<>) = (++)
```

Но также и например:

```haskell
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> b       = b
    a       <> Nothing = a
    Just a  <> Just b  = Just (a <> b)
```

Ещё больше интересных стандартных классов типов

Моноид добавляет ещё нейтральный элемент:

```
class Semigroup a => Monoid a where
  -- | Identity of 'mappend'
  mempty  :: a
```

Кстати, пример на ограничения в классе.

Ещё больше интересных стандартных классов типов

Моноид добавляет ещё нейтральный элемент:

```
class Semigroup a => Monoid a where
  -- | Identity of 'mappend'
  mempty  :: a
```

Кстати, пример на ограничения в классе. Для списков всё очевидно:

```
instance Monoid [a] where
  mempty  = []
```

Ещё больше интересных стандартных классов типов

Моноид добавляет ещё нейтральный элемент:

```
class Semigroup a => Monoid a where
  -- | Identity of 'mappend'
  mempty  :: a
```

Кстати, пример на ограничения в классе. Для списков всё очевидно:

```
instance Monoid [a] where
  mempty  = []
```

А вот это интересный случай:

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

Множество экземпляров класса для одного типа

Для числовых типов моноид не один: может быть умножение с единицей, сложение с нулём... что делать?

В стандартной библиотеке нет `instance Monoid` для числовых типов. Они есть для обёрток (в модуле `Data.Monoid`):

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

`newtype` — сильно ограниченная форма `data`, позволяющая только такие «обёртки» вокруг одного поля. Смысл в эффективности (во время исполнения `newtype` полностью стирается). При этом в отличие от `type` синонимов, это именно отдельные типы с точки зрения системы типов.

Множество экземпляров класса для одного типа

Для числовых типов моноид не один: может быть умножение с единицей, сложение с нулём... что делать?

В стандартной библиотеке нет `instance Monoid` для числовых типов. Они есть для обёрток (в модуле `Data.Monoid`):

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

`newtype` — сильно ограниченная форма `data`, позволяющая только такие «обёртки» вокруг одного поля. Смысл в эффективности (во время исполнения `newtype` полностью стирается). При этом в отличие от `type` синонимов, это именно отдельные типы с точки зрения системы типов.

```
Prelude Data.Monoid> Product 1 <> Product 2 <> Product 3 <> Product 4
Product {getProduct = 24}
Prelude Data.Monoid> getSum $ mconcat $ map Sum [1..10]
55
```

(`mconcat` сворачивает список операцией моноида.)

И ещё больше классов типов

Последний на сегодня у нас функтор! Функторами являются типы, «по которым можно делать `map`»:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b   -- синоним: оператор (<$>)
```

И ещё больше классов типов

Последний на сегодня у нас функтор! Функторами являются типы, «по которым можно делать map»:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b   -- синоним: оператор (<$>)
```

Очевидно, списки:

```
Prelude> (* 2) <$> [1..5] -- то же, что map (* 2) [1..5]
[2,4,6,8,10]
```

И ещё больше классов типов

Последний на сегодня у нас функтор! Функторами являются типы, «по которым можно делать map»:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b   -- синоним: оператор (<$>)
```

Очевидно, списки:

```
Prelude> (* 2) <$> [1..5] -- то же, что map (* 2) [1..5]
[2,4,6,8,10]
```

Почти очевидно, Maybe:

```
Prelude> (* 2) <$> Just 5
Just 10
```

И ещё больше классов типов

Последний на сегодня у нас функтор! Функторами являются типы, «по которым можно делать map»:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b   -- синоним: оператор (<$>)
```

Очевидно, списки:

```
Prelude> (* 2) <$> [1..5] -- то же, что map (* 2) [1..5]
[2,4,6,8,10]
```

Почти очевидно, Maybe:

```
Prelude> (* 2) <$> Just 5
Just 10
```

Даже кортежи (почему операция применилась к последнему элементу?):

```
Prelude> (* 2) <$> (123, 456)
(123,912)
```

Закон и порядок

Функтор — первый класс типов, у которого есть *законы*, которым должны удовлетворять экземпляры:

- `fmap id = id`
- `fmap (f . g) = fmap f . fmap g`

Это соответствие никак формально не проверяется. ~~Просто за написание экземпляров, не выполняющих эти формулы, принято бить по рукам :)~~

Закон и порядок

Функтор — первый класс типов, у которого есть *законы*, которым должны удовлетворять экземпляры:

- `fmap id = id`
- `fmap (f . g) = fmap f . fmap g`

Это соответствие никак формально не проверяется. ~~Просто за написание экземпляров, не выполняющих эти формулы, принято бить по рукам :)~~

Кстати, мы до сих пор не видели этот оператор — точку. Это композиция функций:

```
Prelude> filter (> 20) . map (* 2) $ [1..15]
[22,24,26,28,30]
```

Её удобно использовать в определениях, в анонимных функциях и т.д.:

```
desort = reverse . sort
Prelude Data.List> desort [11,33,22,44,99,55]
[99,55,44,33,22,11]
```

Спасибо за внимание!