Лекция 3: типы данных

Функциональное программирование на Haskell

Алексей Романов

21 февраля 2018

МИЭТ

Типы-«перечисления»

- Тип `Bool` не встроен в язык, а определён в стандартной библиотеке:

  ```
  data Bool = False | True deriving (...)
  ```

- Ключевое слово `data` начинает определение.
- `Bool` — название типа.
- `False` и `True` называются конструкторами данных.
- Читаем как «У типа `Bool` есть ровно два значения: `False` и `True`.»
- О `deriving` позже.

Типы-«перечисления»

- Тип `Bool` не встроен в язык, а определён в стандартной библиотеке:

  ```
  data Bool = False | True deriving (...)
  ```

- Ключевое слово `data` начинает определение.
- `Bool` — название типа.
- `False` и `True` называются конструкторами данных.
- Читаем как «У типа `Bool` есть ровно два значения: `False` и `True`.»
- Так определяется любой тип с фиксированным перечнем значений (как `enum` в других языках):

  ```
  data Weekday = Monday | Tuesday | ...
  data FileMode = Read | Write | Append | ...
  ```

- Какие образцы у этих типов (кроме переменных и `_`)?

Типы-«перечисления»

- Тип `Bool` не встроен в язык, а определён в стандартной библиотеке:

  ```
  data Bool = False | True deriving (...)
  ```

- Ключевое слово `data` начинает определение.
- `Bool` — название типа.
- `False` и `True` называются конструкторами данных.
- Читаем как «У типа `Bool` есть ровно два значения: `False` и `True`.»
- Так определяется любой тип с фиксированным перечнем значений (как `enum` в других языках):

  ```
  data Weekday = Monday | Tuesday | ...
  data FileMode = Read | Write | Append | ...
  ```

- Какие образцы у этих типов (кроме переменных и `_`)?
- Каждое значение (конструктор) — образец.

Типы-«структуры»

- `data` также используется для аналогов структур: типов с несколькими полями. Например,

  ```
  data Point = Point Double Double
  ```

- Первое `Point` — название типа.
- Второе — конструктора.
- Когда конструктор один, названия обычно совпадают. Всегда по контексту можно определить, что из них имеется в виду.
- Образцы этого типа

Типы-«структуры»

- data также используется для аналогов структур: типов с несколькими полями. Например,

```
data Point = Point Double Double
```

- Первое Point — название типа.
- Второе — конструктора.
- Когда конструктор один, названия обычно совпадают. Всегда по контексту можно определить, что из них имеется в виду.
- Образцы этого типа это

```
Point образец_Double образец_Double
```

- Зададим значение типа Point и функцию на них:

```
Prelude> let origin = Point 0 0

Prelude> let xCoord (Point x _) = x
Prelude> xCoord origin
0.0
```

Типы-«структуры»

- Конструктор является функцией

```
Prelude> :t Point
```

Типы-«структуры»

- Конструктор является функцией

```
Prelude> :t Point
Point :: Double -> Double -> Point
```

- Мы можем дать полям конструктора имена. Это:
  - Документирует их смысл.
  - Определяет функции, возвращающие их значения.

```
Prelude> data Point = Point { xCoord ::
   Double, yCoord :: Double } deriving Show
Prelude> :t yCoord
yCoord2 :: Point -> Double
Prelude> yCoord (Point 0 (-1))
-1.0
```

- Особенно полезно, когда полей много, и только по типу не понять, какое где.
- Если успеем, вернёмся к записям в конце лекции.

▸ Синтаксис записей

Алгебраические типы: общий случай

- Может быть несколько конструкторов с полями:

```
data IpAddress = IPv4 String | IPv6 String
```

- Создание значения:

```
Prelude> let googleDns = IPv4 "8.8.8.8"
```

- Образцы: `IPv4 образец_String` и `IPv6 образец_String`.
- Функции часто определяются по уравнению на каждый конструктор:

```
asString (IPv4 ip4) = ip4
asString (IPv6 ip6) = ip6
```

Алгебраические типы: общий случай

- Может быть несколько конструкторов с полями:

```
data IpAddress = IPv4 String | IPv6 String
```

- Создание значения:

```
Prelude> let googleDns = IPv4 "8.8.8.8"
```

- Образцы: `IPv4 образец_String` и `IPv6 образец_String`.
- Функции часто определяются по уравнению на каждый конструктор:

```
asString (IPv4 ip4) = ip4
asString (IPv6 ip6) = ip6
```

- Но не обязательно:

```
isLocalhost (IPv4 "127.0.0.1") = True
isLocalhost (IPv6 "0:0:0:0:0:0:0:1") = True
isLocalhost _ = False
```

Алгебраические типы: общий случай

- Может быть несколько конструкторов с полями:

```
data IpAddress = IPv4 String | IPv6 String
```

- Создание значения:

```
Prelude> let googleDns = IPv4 "8.8.8.8"
```

- Образцы: `IPv4 образец_String` и `IPv6 образец_String`.
- Функции часто определяются по уравнению на каждый конструктор:

```
asString (IPv4 ip4) = ip4
asString (IPv6 ip6) = ip6
```

- Но не обязательно:

```
isLocalhost (IPv4 "127.0.0.1") = True
isLocalhost (IPv6 "0:0:0:0:0:0:0:1") = True
isLocalhost _ = False
```

- Ещё пример: `data ImageFormat = JPG | PNG | IF String`

Типы с параметрами (полиморфные)

- В определении типа могут быть параметры.
- Пример:

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

- `Maybe` — конструктор типа.
- Вместо a можно подставить любой тип, и получить снова тип: `Maybe Int`, `Maybe Bool`.
- Примеры их значений: `Just 1`, `Nothing`.
- В Haskell нет отношения «Надтип — подтип», вместо него «Более общий — частный случай».
- `Maybe a` описывает необязательное значение типа a.

Типы с параметрами (полиморфные)

- В определении типа могут быть параметры.
- Пример:

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

- `Maybe` — конструктор типа.
- Вместо `a` можно подставить любой тип, и получить снова тип: `Maybe Int`, `Maybe Bool`.
- Примеры их значений: `Just 1`, `Nothing`.
- В Haskell нет отношения «Надтип — подтип», вместо него «Более общий — частный случай».
- `Maybe a` описывает необязательное значение типа `a`.
- Ещё часто встречающийся тип:

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

- Могут быть сколь угодно сложные комбинации.
- Например, `Either (Either Int Char) (Maybe Bool)` со значением `Right Nothing`.

`Maybe` и `null`

- `Nothing` похоже на `null` в C-подобных языках, но явно прописано в типах.
- Не нужно для каждой функции документировать, как работает с `null`, достаточно посмотреть на наличие `Maybe` в сигнатуре.
- И поэтому нет возможного несовпадения реализации с документацией.
- Нет типов, у которых `null` нет и с отсутствием значения приходится что-то придумывать.
- Есть `Maybe (Maybe a)` (бывает полезно).

`Maybe` и `null`

- `Nothing` похоже на `null` в C-подобных языках, но явно прописано в типах.
- Не нужно для каждой функции документировать, как работает с `null`, достаточно посмотреть на наличие `Maybe` в сигнатуре.
- И поэтому нет возможного несовпадения реализации с документацией.
- Нет типов, у которых `null` нет и с отсутствием значения приходится что-то придумывать.
- Есть `Maybe (Maybe a)` (бывает полезно).
- *«`null` — моя ошибка ценой в миллиард долларов... Я планировал сделать любое использование ссылок [в Algol] абсолютно безопасным. Но не удержался от соблазна добавить `null` просто потому, что его было так легко реализовать.»* (Тони Хоар, QCon 2009)

Кортежи

- Мы могли бы определить типы

  ```
  data Pair a b = Pair a b
  data Triple a b c = Triple a b c
  ...
  ```

  (заметьте разницу смыслов `Pair`, `a` и `b` в левой и правой частях!)
- Это универсальные произведения.

Кортежи

- Мы могли бы определить типы

  ```
  data Pair a b = Pair a b
  data Triple a b c = Triple a b c
  ...
  ```

  (заметьте разницу смыслов `Pair`, `a` и `b` в левой и правой частях!)
- Это универсальные произведения.
- Но определять их не нужно, так как такие типы уже есть, со специальным синтаксисом:
  `(a, b)` (или `(,) a b`),
  `(a, b, c)` (или `(,,) a b c`) и т.д.
- Синтаксис значений и образцов для них аналогичный:
  `(True, Just 'a') ::`

Кортежи

- Мы могли бы определить типы

  ```
  data Pair a b = Pair a b
  data Triple a b c = Triple a b c
  ...
  ```

  (заметьте разницу смыслов `Pair`, `a` и `b` в левой и правой частях!)
- Это универсальные произведения.
- Но определять их не нужно, так как такие типы уже есть, со специальным синтаксисом:
  `(a, b)` (или `(,) a b`),
  `(a, b, c)` (или `(,,) a b c`) и т.д.
- Синтаксис значений и образцов для них аналогичный:
  `(True, Just 'a') :: (Bool, Maybe Char)`.
- Максимальный размер 62.

Рекурсивные типы и списки

- Тип может быть рекурсивным, то есть в правой части используется тот тип, который мы определяем.
- Самый важный пример такого типа: списки. Мы могли бы их определить как

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

  Вместо этого они имеют специальный синтаксис, как и кортежи, как будто их определение

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

- Любой `[a]` или пуст, или имеет *голову* типа `a` и *хвост* типа `[a]`.
- `:` правоассоциативно: `1 : 2 : 4 : []` читается `1 : (2 : (4 : []))`.
- Сокращение (относится и к значениям и к образцам): `[x, y, z]` означает `x : y : z : []`, а `[x]` — `x : []`.

Синонимы типов

- На этом с `data` разобрались (пока что).

Синонимы типов

- На этом с `data` разобрались (пока что).
- Есть ещё два вида определения типов.
- `type` объявляет синоним типа — другое имя для существующего типа.
- В стандартной библиотеке:

```
type String = [Char]
type FilePath = String
```

- Оба определения теперь считаются плохими! Для `String` подробности в будущих лекциях.

Синонимы типов

- На этом с `data` разобрались (пока что).
- Есть ещё два вида определения типов.
- `type` объявляет синоним типа — другое имя для существующего типа.
- В стандартной библиотеке:

```
type String = [Char]
type FilePath = String
```

- Оба определения теперь считаются плохими! Для `String` подробности в будущих лекциях.
- Для `FilePath` дело в том, что осмысленные операции над путями не те, что над строками:

```
Prelude Data.List> isPrefixOf
    ("C:\\Program␣Files" :: FilePath)
    ("C:\\Program␣Files␣(x86)" :: FilePath)
True
```

newtype

- newtype как раз позволяет определить тип с тем же представлением, что существующий, но с другими операциями. Т.е. разумнее было бы

  ```
  newtype FilePath = FilePath String
  ```

- Или `FilePath [String]` со списком директорий.
- Ещё можно рассмотреть

  ```
  newtype EMail = EMail String
  ```

- У type конструкторов значений нет, у newtype — всегда один с одним полем.
- Между newtype и data с одним конструктором с одним полем есть разница из-за ленивости.
- Пока из них предпочитаем первое.

Полиморфные функции и значения

- Какой тип функции

```
Prelude> let id x = x
Prelude> :t id
```

Полиморфные функции и значения

- Какой тип функции

```
Prelude> let id x = x
Prelude> :t id
id :: a -> a
```

- Более явно: `id :: forall a. a -> a`.
  - Чтобы написать это в коде, нужно `ExplicitForAll`.
- Аналогично

```
Prelude> let foo (_, x, y) = (y, x, x)
Prelude> :t foo
```

Полиморфные функции и значения

- Какой тип функции

```
Prelude> let id x = x
Prelude> :t id
id :: a -> a
```

- Более явно: `id :: forall a. a -> a`.
  - Чтобы написать это в коде, нужно `ExplicitForAll`.
- Аналогично

```
Prelude> let foo (_, x, y) = (y, x, x)
Prelude> :t foo
id :: (a1, c, a2) -> (a2, c, c)
-- или (a, b, c) -> (c, b, b)
```

- `Just :: a -> Maybe a`
- И `[] :: [a]` — полиморфные значения не обязательно функции.

Полезные расширения для полиморфных функций

- Параметр типа можно передать явно:

```
Prelude> :set -XTypeApplications
Prelude> :t id @Int
id @Int :: Int -> Int
```

Полезные расширения для полиморфных функций

- Параметр типа можно передать явно:

```
Prelude> :set -XTypeApplications
Prelude> :t id @Int
id @Int :: Int -> Int
```

- По умолчанию область видимости переменной типа — сигнатура, где она появилась. Например, в

```
g :: [a] -> [a]
g (x:xs) = xs ++ [ x :: a ]
```

Полезные расширения для полиморфных функций

- Параметр типа можно передать явно:

```
Prelude> :set -XTypeApplications
Prelude> :t id @Int
id @Int :: Int -> Int
```

- По умолчанию область видимости переменной типа — сигнатура, где она появилась. Например, в

```
g :: [a] -> [a]
g (x:xs) = xs ++ [ x :: a ]
```

две переменные a — разные и читаются как

```
g :: forall a. [a] -> [a]
g (x:xs) = xs ++ [ x :: forall a. a ]
```

Полезные расширения для полиморфных функций

- Параметр типа можно передать явно:

```
Prelude> :set -XTypeApplications
Prelude> :t id @Int
id @Int :: Int -> Int
```

- По умолчанию область видимости переменной типа — сигнатура, где она появилась. Например, в

```
g :: [a] -> [a]
g (x:xs) = xs ++ [ x :: a ]
```

две переменные a — разные и читаются как

```
g :: forall a. [a] -> [a]
g (x:xs) = xs ++ [ x :: forall a. a ]
```

(так что функция не скомпилируется).

Область видимости переменных типа

- Расширение `ScopedTypeVariables` расширяет область видимости, но только для переменных с явным `forall`.
- С ним пример с прошлого слайда можно записать как

```
g :: forall a. [a] -> [a]
g (x:xs) = xs ++ [ x :: a ]
```

- Можно также вводить новые переменные сигнатурой для выражения *внутри* определения, они будут видимы в этом выражении.
- Точные правила можно найти в документации, в разделе Lexically scoped type variables.

Структурно-рекурсивные функции

- Функции над рекурсивными типами часто тоже рекурсивны (или вызывают рекурсивные).

```
length :: [a] -> Int
length [] = 0
length (_ : xs) = length xs + 1
```

- Натуральные числа — «почётный» рекурсивный тип.

Структурно-рекурсивные функции

- Функции над рекурсивными типами часто тоже рекурсивны (или вызывают рекурсивные).

```
length :: [a] -> Int
length [] = 0
length (_ : xs) = length xs + 1
```

- Натуральные числа — «почётный» рекурсивный тип.
- Натуральное число это либо 0, либо $n + 1$, где n — натуральное число.
- Соответственно, функции над ними тоже определяются рекурсивно:

```
replicate _ 0 = []
replicate x n = x : replicate x (n - 1)
```

- Раньше `n + 1` (и вообще `+ константа`) было образцом, в Haskell 2010 их исключили.

Алгебраические типы

- Несколько полей у конструктора соответствуют операции декартова произведения в теории множеств.

- А несколько конструкторов?

Алгебраические типы

- Несколько полей у конструктора соответствуют операции декартова произведения в теории множеств.

- А несколько конструкторов операции дизъюнктного объединения $A_1 \sqcup A_2 \sqcup \ldots = \{1\} \times A_1 \cup \{2\} \times A_2 \cup \ldots$

- Сравните помеченное объединение с непомеченным `union` в C/C++.
- И с `boost::variant` (`std::variant` в C++17).

Алгебраические типы

- Несколько полей у конструктора соответствуют операции декартова произведения в теории множеств.
  - В логике конъюнкции, а в алгебре произведению.
- А несколько конструкторов операции дизъюнктного объединения $A_1 \sqcup A_2 \sqcup \ldots = \{1\} \times A_1 \cup \{2\} \times A_2 \cup \ldots$
  - В логике дизъюнкции, а в алгебре сумме.
- Сравните помеченное объединение с непомеченным `union` в C/C++.
- И с `boost::variant` (`std::variant` в C++17).
- Тип функций — множество функций в теории множеств, импликация в логике и возведение в степень в алгебре.

Алгебраические типы

- Несколько полей у конструктора соответствуют операции декартова произведения в теории множеств.
  - В логике конъюнкции, а в алгебре произведению.
- А несколько конструкторов операции дизъюнктного объединения $A_1 \sqcup A_2 \sqcup \ldots = \{1\} \times A_1 \cup \{2\} \times A_2 \cup \ldots$
  - В логике дизъюнкции, а в алгебре сумме.
- Сравните помеченное объединение с непомеченным `union` в C/C++.
- И с `boost::variant` (`std::variant` в C++17).
- Тип функций — множество функций в теории множеств, импликация в логике и возведение в степень в алгебре.
- Обычные законы алгебры выполняются для типов с точностью до изоморфизма, как и для множеств.

Типы и модули

- Типы можно импортировать и экспортировать.
- `НазваниеТипа(..)` в списке экспорта указывает, что конструкторы тоже включены в список.
- Просто НазваниеТипа экспортирует только тип без конструкторов.
- Можно также перечислить явно, какие именно конструкторы экспортируются.
- Для импорта всё аналогично.

Записи

К определению записей

- ```
  Prelude> let aPoint = Point 0 (-1)
  Prelude> aPoint { xCoord = 1 }
  ```
  ```

Записи

К определению записей

- ```
  Prelude> let aPoint = Point 0 (-1)
  Prelude> aPoint { xCoord = 1 }
  Point {xCoord = 1.0, yCoord = -1.0}
  Prelude> aPoint
  ```
  ```

Записи

К определению записей

- ```
  Prelude> let aPoint = Point 0 (-1)
  Prelude> aPoint { xCoord = 1 }
  Point {xCoord = 1.0, yCoord = -1.0}
  Prelude> aPoint
  Point {xCoord = 0.0, yCoord = -1.0}
  ```
  ```

Записи

К определению записей

- ```
  Prelude> let aPoint = Point 0 (-1)
  Prelude> aPoint { xCoord = 1 }
  Point {xCoord = 1.0, yCoord = -1.0}
  Prelude> aPoint
  Point {xCoord = 0.0, yCoord = -1.0}
  Prelude> let Point {xCoord = x'} = it
  Prelude> x'
  ```
  ```

Записи

К определению записей

- ```
  Prelude> let aPoint = Point 0 (-1)
  Prelude> aPoint { xCoord = 1 }
  Point {xCoord = 1.0, yCoord = -1.0}
  Prelude> aPoint
  Point {xCoord = 0.0, yCoord = -1.0}
  Prelude> let Point {xCoord = x'} = it
  Prelude> x'
  0.0
  ```
  ```

Записи

К определению записей

- ```
  Prelude> let aPoint = Point 0 (-1)
  Prelude> aPoint { xCoord = 1 }
  Point {xCoord = 1.0, yCoord = -1.0}
  Prelude> aPoint
  Point {xCoord = 0.0, yCoord = -1.0}
  Prelude> let Point {xCoord = x'} = it
  Prelude> x'
  0.0

  Prelude> :set -XNamedFieldPuns -XRecordWildCards
  Prelude> let f (Point { xCoord, yCoord }) = xCoord +
      yCoord
  Prelude> let g (Point { .. }) = xCoord + yCoord
  ```
- Два последних образца — сокращения

  ```
  Point { xCoord = xCoord, yCoord = yCoord }.
  ```
  ```

Записи: минусы

К определению записей

- `Prelude> Point { xCoord = 0 }`

Записи: минусы

◂ К определению записей

- `Prelude> Point { xCoord = 0 }`

  без `yCoord` компилируется только с предупреждением (можно сделать ошибкой с помощью опций компилятора).

Записи: минусы

◂ К определению записей

- `Prelude> Point { xCoord = 0 }`

  без `yCoord` компилируется только с предупреждением (можно сделать ошибкой с помощью опций компилятора).
- Записи можно применять для типов с несколькими конструкторами, но это создаёт частичные (не всюду определённые) функции.

Записи: минусы

◂ К определению записей

- `Prelude> Point { xCoord = 0 }`

  без `yCoord` компилируется только с предупреждением (можно сделать ошибкой с помощью опций компилятора).
- Записи можно применять для типов с несколькими конструкторами, но это создаёт частичные (не всюду определённые) функции.
- Одинаковые названия полей у двух записей ведут к проблемам.

Записи: минусы

◂ К определению записей

- `Prelude> Point { xCoord = 0 }`

  без `yCoord` компилируется только с предупреждением (можно сделать ошибкой с помощью опций компилятора).
- Записи можно применять для типов с несколькими конструкторами, но это создаёт частичные (не всюду определённые) функции.
- Одинаковые названия полей у двух записей ведут к проблемам.
- Получается две функции с одинаковыми названиями.
- `DisambiguateRecordFields` и `DuplicateRecordFields` улучшают ситуацию.

Записи: минусы

К определению записей

- `Prelude> Point { xCoord = 0 }`

  без `yCoord` компилируется только с предупреждением (можно сделать ошибкой с помощью опций компилятора).
- Записи можно применять для типов с несколькими конструкторами, но это создаёт частичные (не всюду определённые) функции.
- Одинаковые названия полей у двух записей ведут к проблемам.
- Получается две функции с одинаковыми названиями.
- `DisambiguateRecordFields` и `DuplicateRecordFields` улучшают ситуацию.
- Есть ещё решения на уровне библиотек.