

# Типы и пользовательские типы данных

Косарев Дмитрий а.к.а. Kakadu

матмех СПбГУ

12 сентября 2019 г.

## В этих слайдах

- Синтаксис *алгебраических типов данных*
- Несколько примеров того, как превратить предметную область в описание типов

N.B. Программирование на типизированных функциональных языках обычно начинается с описаний типов.

Вопрос к залу: что такое тип?

## Вопрос к залу: что такое тип?

Тип  $T$  у значения с именем  $x$  – это множество совокупность значений, которые могут быть у  $x$ .

Если  $x$  принадлежит типу  $T$ , то  $T$  определяет, какие значения может принимать  $x$ .

В Haskell нотация  $x :: A$ , обозначает, что значение  $x$  принадлежит типу  $A$ , или что  $x$  можно протипизировать типом  $A$ .

Если  $x :: T$ , то говорят, что тип  $T$  *населен* иксом.

Если  $\nexists x$ , таких что  $x :: T$ , то тип  $T$  *не населен*.

# Тип функции

Тип функции, действующей из аргумента типа **A** и возвращающей результат **B**, обозначается как **A→B**.

# Тип функции

Тип функции, действующей из аргумента типа  $A$  и возвращающей результат  $B$ , обозначается как  $A \rightarrow B$ .

Функции от  $n$  аргументов ( $n > 1$ ) моделируются как функции, возвращающие функцию от  $(n - 1)$  аргументов. Например,  $A \rightarrow (B \rightarrow C)$ .

# Тип функции

Тип функции, действующей из аргумента типа  $A$  и возвращающей результат  $B$ , обозначается как  $A \rightarrow B$ .

Функции от  $n$  аргументов ( $n > 1$ ) моделируются как функции, возвращающие функцию от  $(n - 1)$  аргументов. Например,  $A \rightarrow (B \rightarrow C)$ .

Ассоциативность правая: т.е.  $A \rightarrow (B \rightarrow C)$  — это тот же самый тип, что и  $A \rightarrow B \rightarrow C$ .

# Каррирование (currying)

В “обычных” языках все аргументы функции передаются сразу (передается n-ка аргументов).

$$(A, B, C, D) \rightarrow R$$

В ФП принято передавать аргументы по одному, т.е. функция принимает один аргумент и возвращает функцию, которая принимает другой аргумент, и т.д.

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow R$$



# Параметрический полиморфизм

*Параметрический полиморфизм* – это такое свойство участка кода, что он может выполняться для разных типов (в Java/C# называется generics).

# Параметрический полиморфизм

*Параметрический полиморфизм* – это такое свойство участка кода, что он может выполняться для разных типов (в Java/C# называется generics).

*Типовые переменные* в типах полиморфных функций пишутся с маленькой буквы. Например,  $a \rightarrow b$  или  $a \rightarrow b \rightarrow c$  или  $a \rightarrow (a \rightarrow b) \rightarrow b$

# Параметрический полиморфизм

*Параметрический полиморфизм* – это такое свойство участка кода, что он может выполняться для разных типов (в Java/C# называется generics).

*Типовые переменные* в типах полиморфных функций пишутся с маленькой буквы. Например,  $a \rightarrow b$  или  $a \rightarrow b \rightarrow c$  или  $a \rightarrow (a \rightarrow b) \rightarrow b$

*Имена конкретных типов* пишутся с заглавной буквы. Например, **Int**, **String**, **Float**.

# Параметрический полиморфизм

*Параметрический полиморфизм* – это такое свойство участка кода, что он может выполняться для разных типов (в Java/C# называется generics).

*Типовые переменные* в типах полиморфных функций пишутся с маленькой буквы. Например,  $a \rightarrow b$  или  $a \rightarrow b \rightarrow c$  или  $a \rightarrow (a \rightarrow b) \rightarrow b$

*Имена конкретных типов* пишутся с заглавной буквы. Например, **Int**, **String**, **Float**.  
Параметры типа (конкретные или типовые переменные) обычно пишутся справа от имени типа: **Maybe**  $a$  или **Map**  $\text{key value}$ .

# Параметрический полиморфизм

*Параметрический полиморфизм* – это такое свойство участка кода, что он может выполняться для разных типов (в Java/C# называется generics).

*Типовые переменные* в типах полиморфных функций пишутся с маленькой буквы. Например,  $a \rightarrow b$  или  $a \rightarrow b \rightarrow c$  или  $a \rightarrow (a \rightarrow b) \rightarrow b$

*Имена конкретных типов* пишутся с заглавной буквы. Например, **Int**, **String**, **Float**. Параметры типа (конкретные или типовые переменные) обычно пишутся справа от имени типа: **Maybe**  $a$  или **Map**  $key\ value$ .

Для типа *списков* значений типа **T** есть специальный синтаксис: **[T]**.

Итого функция  $[a] \rightarrow (a \rightarrow b) \rightarrow [b]$  принимает список значений типа  $[a]$ , затем функцию  $a \rightarrow b$  и возвращает  $[b]$ .

Если есть полиморфный тип, то его тип можно “уточнить” (мономорфизировать), подставив конкретные типы вместо типовых переменных. Например,

- $[a] \rightarrow (a \rightarrow b) \rightarrow [b]$
- Подставим  $b$  вместо  $a$
- $[b] \rightarrow (b \rightarrow b) \rightarrow [b]$
- Подставим  $\text{Int}$  вместо  $b$
- $[\text{Int}] \rightarrow (\text{Int} \rightarrow \text{Int}) \rightarrow [\text{Int}]$
- Дан тип конечного отображения  $\text{Map key value}$
- Пускай ключами будет  $\text{Int}$
- $\text{Map Int value}$
- а значениями другие конечные отображения из строк...
- $\text{Map Int (Map String v)}$
- ...в  $\text{Bool}$
- $\text{Map Int (Map String Bool)}$

# $\alpha$ -эквивалентные полиморфные типы

## Определение

Два полиморфных типа  $\sigma$  и  $\tau$  эквивалентны, если можно так переименовать типовые переменные в  $\sigma$ , чтобы получить  $\tau$ ; и наоборот.

Пример 1. Типы  $a \rightarrow a$  и  $b \rightarrow b$  эквивалентны.

Пример 2. Типы  $a \rightarrow (a \rightarrow b) \rightarrow b$  и  $a \rightarrow (a \rightarrow a) \rightarrow a$  не эквивалентны, так как никаким переименованием из правого типа не получить левый.

# Алгебраические типы данных

## В синтаксисе Haskell

```
data TypeName arg1 arg2 ... argk =  
    C1 t11 t12 ... t1n1  
  | C2 t21 t22 ... t2n2  
  | ...  
  | Cm tm1 tm2 ... tmnm
```

$C_i$  – конструкторы для типа **TypeName**  
 $arg_i$  – типовые параметры типа **TypeName**  
 $t_{ij}$  – типы-аргументы конструктора  $C_i$

## Примеры нерекурсивных типов

- `data Bool = True | False`
- `data Status1 = On | Off`
- `data Maybe a = Just a | Nothing`
- `data Either a b = Left a | Right b`

Значения можно *конструировать*.

- `True :: Bool`
- `Just :: Maybe a`
- `Just True :: Maybe Bool`



# Натуральные числа в стиле Пеано

Наверное, самый простой рекурсивный тип.

Положим у нас есть “ноль” (ну или “один”) и есть “следующий”.

# Натуральные числа в стиле Пеано

Наверное, самый простой рекурсивный тип.

Положим у нас есть “ноль” (ну или “один”) и есть “следующий”.

```
data Nat = Zero | Succ Nat
```

```
Zero :: Nat
```

```
Succ Zero :: Nat
```

```
Succ (Succ Zero) :: Nat
```

# Натуральные числа в стиле Пеано

Наверное, самый простой рекурсивный тип.

Положим у нас есть “ноль” (ну или “один”) и есть “следующий”.

```
data Nat = Zero | Succ Nat
```

```
Zero :: Nat
```

```
Succ Zero :: Nat
```

```
Succ (Succ Zero) :: Nat
```

Упражнение(простое): написать функцию для сложения и умножения пары чисел Пеано.

# Связный список – пример рекурсивного алгебраического типа

? Что такое *связный список*?

# Связный список – пример рекурсивного алгебраического типа

? Что такое *связный список*?

? А что такое *список* вообще?

# Связный список – пример рекурсивного алгебраического типа

? Что такое *связный список*?

? А что такое *список* вообще?

Ещё примеры:

- `data [] a = [] | a : [a]`
- `data List a = Nil | Cons a (List a)`
- `data ListG a r = Nil | Cons a r`
- `data Fix f = Fix (f (Fix f))`

Упражнение (сложное): из типов `ListG` и `Fix` можно соорудить что-то похожее на стандартный список Haskell. Предъявите биекцию между типами (например, с помощью двух функций).

# JSON

- Числа
- Строки
- Массивы
- Объекты как набор пар “ключ-значение”

# JSON

- Числа
- Строки
- Массивы
- Объекты как набор пар “ключ-значение”

```
data JSON = JNull
          | JBool Bool
          | JNum   Float
          | JStr   String
          | JArray [JSON]
          | JObject [(String, JSON)]
```



## Пример про почту (1/2)

```
data Contact = Contact
  { name :: Name
  , emailContactInfo :: EmailContactInfo
  , postalContactInfo :: PostalContactInfo }
```

## Пример про почту (1/2)

```
data Contact = Contact
  { name :: Name
  , emailContactInfo :: EmailContactInfo
  , postalContactInfo :: PostalContactInfo }
```

Хочется, чтобы у контакта был *хотя бы один* адрес: либо электронной, либо физической почты.

## Пример про почту (1/2)

```
data Contact = Contact
  { name :: Name
  , emailContactInfo :: EmailContactInfo
  , postalContactInfo :: PostalContactInfo }
```

Хочется, чтобы у контакта был *хотя бы один* адрес: либо электронной, либо физической почты.

Что вы думаете о вот таком?

```
data Contact = Contact
  { name :: Name
  , emailContactInfo :: Maybe EmailContactInfo
  , postalContactInfo :: Maybe PostalContactInfo }
```

## Пример про почту (2/2)

```
data ContactInfo =  
    EmailOnly      EmailContactInfo  
  | PostOnly      PostalContactInfo  
  | EmailAndPost  EmailContactInfo PostalContactInfo  
  
data Contact = Contact  
  { name          :: Name  
  , contactInfo   :: ContactInfo  
  }
```

Если, посмотрев на тип, сразу понятно какие состояния корректные, а какие нет, то это считается хорошим дизайном.

Пример взят [отсюда](#).

## Содержательный пример (1/3) из [1]

```
import Data.Word

data InetAddr = InetAddr Word8 Word8 Word8 Word8

data ConnectionState = Connecting | Connected | Disconnected

data ConnectionInfo = CInfo
  { state :: ConnectionState
  , server :: InetAddr
  , last_ping_time :: Maybe Time
  , last_ping_id :: Maybe Int
  , session_id :: Maybe String
  , when_initiated :: Maybe Time
  , when_disconnected :: Maybe Time
  }
```

## Содержательный пример (2/3)

```
data Connecting    = Connecting { when_initiated :: Time }
data Disconnected = Disconnected { when_disconnected :: Time }

data Connected    = Connected
  { last_ping    :: Maybe (Time, Int)
  , session_id  :: String }

data ConnectionState =
  SConnecting    Connecting
| SConnected     Connected
| SDisconnected Disconnected
```

## Содержательный пример (3/3)

```
data ConnectionState =  
    SConnecting    Connecting  
| SConnected      Connected  
| SDisconnected   Disconnected
```

```
data ConnectionInfo = Cinfo  
{ state :: ConnectionState  
  , server :: InetAddr }
```

## Содержательный пример (3/3)

```
data ConnectionState =  
    SConnecting    Connecting  
| SConnected      Connected  
| SDisconnected   Disconnected
```

```
data ConnectionInfo = Cinfo  
{ state :: ConnectionState  
  , server :: InetAddr }
```

Лозунг: “плохие” состояния (значения) должны быть непредставимы в типах.



## Простой пример: арифметика (1/2)

```
data Expr =  
    EConst Int  
  | EMul Expr Expr  
  | EAdd Expr Expr
```

Хотим написать функцию

```
eval :: Expr -> Int
```

Как писать функции, которые принимают алгебраические типы?

# Pattern matching a.k.a. сопоставление с образцом

```
data Expr =  
    EConst Int  
  | EMul Expr Expr  
  | EAdd Expr Expr
```

```
eval (EConst n) = ...  
eval (EAdd l r) = ...  
eval (EMul l r) = ...
```

Здесь имена `n`, `l` и `r` задаются программистом.

С ключом компилятора `-Wincomplete-patterns` тот даже сообщит, если некоторые конструкторы не были рассмотрены.

```
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
```

## Простой пример: арифметика (2/2)

```
data Expr =  
    EConst Int  
  | EMul Expr Expr  
  | EAdd Expr Expr
```

Хотим написать функцию

```
eval :: Expr -> Int
```

## Простой пример: арифметика (2/2)

```
data Expr =  
    EConst Int  
  | EMul Expr Expr  
  | EAdd Expr Expr
```

Хотим написать функцию

```
eval :: Expr -> Int
```

Вот решение

```
eval (EConst n) = n  
eval (EAdd l r) = (eval l) + (eval r)  
eval (EMul l r) = (eval l) * (eval r)
```

# Арифметика с переменными

```
data Expr =  
    EConst Int  
  | EMul Expr Expr  
  | EAdd Expr Expr  
  | EVar String      -- extra constructor
```

# Арифметика с переменными

```
data Expr =  
    EConst Int  
  | EMul Expr Expr  
  | EAdd Expr Expr  
  | EVar String      -- extra constructor  
  
lookup :: Eq a => a -> [(a, b)] -> Maybe b  -- from stdlib  
eval   :: [(String, Int)] -> Expr -> Maybe Int
```

# Арифметика с переменными

```
data Expr =  
    EConst Int  
  | EMul Expr Expr  
  | EAdd Expr Expr  
  | EVar String      -- extra constructor  
  
lookup :: Eq a => a -> [(a, b)] -> Maybe b  -- from stdlib  
eval   :: [(String, Int)] -> Expr -> Maybe Int  
  
eval _ (EConst n) = Just n  
eval env (EAdd l r) = Just ((eval env l) + (eval env r))  
eval env (EMul l r) = Just ((eval env l) * (eval env r))  
eval env (EVar s)   = lookup s env
```

## Как не надо делать алгебраические типы?

Мы хотим сделать тип для представления множеств

```
data Set a = EmptySet
           | Concrete [a]
           | Union (Set a) (Set a)
           | Intersection (Set a) (Set a)
           | ...
```

Вопрос: как правильно представить пустое множество: `EmptySet` или `Concrete []` ?



# Как не надо делать алгебраические типы?

Мы хотим сделать тип для представления множеств

```
data Set a = EmptySet
           | Concrete [a]
           | Union (Set a) (Set a)
           | Intersection (Set a) (Set a)
           | ...
```

Вопрос: как правильно представить пустое множество: `EmptySet` или `Concrete []` ?

Лозунг: стоит стараться избегать того, что одни и те же значения можно представить несколькими различными способами.

# Конец

Дальше только “запасные слайды” и список литературы

# Чистые функции

## Определение

Чистая функция – это

- Детерминированная
- В процессе работы не совершающая “побочных эффектов”

Т.е. запрещены: ввод-вывод, случайные значения, присваивания

Н.В. Это свойство *функции*, а не языка программирования

## Parametricity theorem [2]

### Теорема

*О параметричности. Чистые функции с параметрическим полиморфизмом работают одинаково для всех возможных типов.<sup>a</sup>*

---

<sup>a</sup>Для Haskell верна, для strict-языков с некоторыми оговорками.

```
{-# LANGUAGE ExplicitForAll #-}
```

```
id :: forall a . a -> a
```

```
id x = x
```

# Чем функции в программировании отличаются от математических?

# Чем функции в программировании отличаются от математических?

- Аварийное завершение
- Отсутствие завершения

# Чем функции в программировании отличаются от математических?

- Аварийное завершение
- Отсутствие завершения

Функции / Область	математика	программирование
всегда возвращают результат	функции	тотальные функции
могут не вернуть результат	частичные функции	функции



OCaml for the Masses

*Yaron Minsky*

[ссылка](#)



Theorems for free!

*Philip Wadler*

[ссылка](#)