Типы и пользовательские типы данных

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

13 сентября 2019 г.

Оглавление

1. Теоретическая часть про типы
2. Теоретическая часть про алгебраические типы
3. Более практическая часть: как писать типы
4. Как писать код с использованием алгебраических типов?

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.

Оглавление

1. Теоретическая часть про типы
2. Теоретическая часть про алгебраические типы
3. Более практическая часть: как писать типы
4. Как писать код с использованием алгебраических 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->B`.

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

Тип функции

Тип функции, действующей из аргумента типа `A` и возвращающей результат `B`, обозначается как `A->B`.

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

Ассоциативность правая: т.е. `A -> (B -> C)` – это тот же самый тип, что и `A -> B -> C`.

Каррирование (currying)

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

```
(A,B,C,D) -> 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 -> B -> C -> D -> 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 -> b` или `a -> b -> c` или `a -> (a -> b) -> 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 -> b` или `a -> b -> c` или `a -> (a -> b) -> 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 -> b` или `a -> b -> c` или `a -> (a -> b) -> 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`.

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 -> b` или `a -> b -> c` или `a -> (a -> b) -> 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] -> (a->b) -> [b]` принимает список значений типа `[a]`, затем функцию `a->b` и возвращает `[b]`.

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

- `[a] -> (a->b) -> [b]`
- Подставим `b` вместо `a`
- `[b] -> (b->b) -> [b]`
- Подставим `Int` вместо `b`
- `[Int] -> (Int->Int) -> [Int]`

- Дан тип конечного отображения `Map key value`
- Пускай ключами будет `Int`
- `Map Int value`
- а значениями другие конечные отображения из строк...
- `Map Int (Map String v)`
- ...в `Bool`
- `Map Int (Map String Bool)`

α-эквивалентные полиморфные типы

Определение

Два полиморфных типа σ и τ эквивалентны, если можно так переименовать типовые переменные в σ, чтобы получить τ; и наоборот.

Пример 1. Типы `a -> a` и `b -> b` эквивалентны.

Пример 2. Типы `a -> (a -> b) -> b` и `a -> (a -> a) -> a` не эквивалентны, так как никаким переименованием из правого типа не получить левый.

Оглавление

1. Теоретическая часть про типы
2. Теоретическая часть про алгебраические типы
3. Более практическая часть: как писать типы
4. Как писать код с использованием алгебраических типов?

Алгебраические типы данных (ADT)

В синтаксисе Haskell

$$
\begin{aligned}
&\texttt{data TypeName } arg_1\ arg_2\ \ldots\ arg_k\ \texttt{=} \\
&\quad C_1\ t_{11}\ t_{12}\ \ldots\ t_{1n_1} \\
&\mid C_2\ t_{21}\ t_{22}\ \ldots\ t_{2n_2} \\
&\mid \ldots \\
&\mid C_m\ t_{m1}\ t_{m2}\ \ldots\ t_{mn_m}
\end{aligned}
$$

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`

- $k = 0, m = 2, n_i = 0$
- $k = 0, m = 2, n_i = 0$
- $k = 1, m = 2, n_1 = 1, n_2 = 0$
- $k = 2, m = 2, n_i = 1$

ADT vs. GADT. Конструкторы порождают функции

ADT

```
data TypeName typevar_1 ... typevar_k =
    C_1 t_11 t_12 ... t_1n_1
  | C_2 t_21 t_22 ... t_2n_2
  | ...
  | C_m t_m1 t_m2 ... t_mn_m
```

GADT богаче ADT (G = generalized), т.к. программист может выбирать τ_{ij}
В синтаксисе GADT явно видно, что конструкторы ведут себя как функции.

Синтаксис GADT намекает, что конструкторы можно использовать как функции

```
data TypeName typevar_1 ... typevar_k where
  C_1 :: t_11 -> ... -> t_1n_1 -> TypeName τ_11 ... τ_1k
  C_2 :: t_21 -> ... -> t_2n_2 -> TypeName τ_21 ... τ_2k
  ...
  C_m :: t_m1 -> ... -> t_mn_m -> TypeName τ_m1 ... τ_mk
```

Вообще, GADT – отдельная сложная тема.

Здесь они появились только для того, чтобы показать,
что конструкторы похожи на функции.

Интересующиеся, могут заглянуть в туториал [3].

Как конструировать значения алгебраических типов?

Только с помощью конструкторов

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

```
Nothing :: Maybe anytype
Just :: a -> Maybe a
Just 1 :: Maybe Int
```

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

```
Left 1 :: Either Int a
Right "Strr" :: Either a String
```

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

```
Nil :: List a
Cons 1 Nil  :: List Int
Cons 2 (Cons 1 Nil) :: List Int
```

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

```
[] :: [a]
["a"] :: [String]
"a":[] :: [String]
["b", "a"] :: [String]
"b":"a":[] :: [String]
```

ADT и объекты

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


-- construction of values
Nothing :: Maybe a
Just 5 :: Maybe Int
```

```csharp
// C#
public abstract class Maybe<T> { }
public class Nothing<T> : Maybe<T> { }
public class Just<T> : Maybe<T>
{
    public T Value { get; set; }
}

(Maybe<T>)(new Nothing)
(Maybe<Integer>)(new Just<Integer>()
                        { Value = 42 })
```

Известный пример: натуральные числа в стиле Пеано

Наверное, самый простой рекурсивный тип.

Символизирует числа в “палочковой” системе исчисления.

Положим у нас есть “ноль” (ну или “один”) и есть “следующий за другим числом Пеано”.

У нас будет два конструктора: `Zero` без аргументов, и `Succ` с одним аргументом.

```
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.

Оглавление

1 Теоретическая часть про типы

2 Теоретическая часть про алгебраические типы

3 Более практическая часть: как писать типы

4 Как писать код с использованием алгебраических типов?

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]
          | JObj   [(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)

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

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

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

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.

Оглавление

1. Теоретическая часть про типы
2. Теоретическая часть про алгебраические типы
3. Более практическая часть: как писать типы
4. Как писать код с использованием алгебраических типов?

Простой пример: арифметика (1/2)

```haskell
data Expr = Const Int
          | Mul Expr Expr
          | Add Expr Expr
```

Хотим написать функцию

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

Как писать функции, которые принимают алгебраические типы?

Pattern matching a.k.a. сопоставление с образцом

```haskell
data Expr = Const Int
          | Mul Expr Expr
          | Add Expr Expr

eval (Const n) = ...
eval (Add l r) = ...
eval (Mul l r) = ...
```

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

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

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

Простой пример: арифметика (2/2)

```
data Expr = Const Int
          | Mul Expr Expr
          | Add Expr Expr
```

Хотим написать функцию

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

Простой пример: арифметика (2/2)

```
data Expr = Const Int
          | Mul Expr Expr
          | Add Expr Expr
```

Хотим написать функцию

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

Вот решение

```
eval (Const n) = n
eval (Add l r) = (eval l) + (eval r)
eval (Mul l r) = (eval l) * (eval r)
```

Pattern matching vs OOP

N.B. В коде на Kotlin могут быть косяки (читал туториал, не компилировал)

```haskell
-- haskell
eval Nothing  = ...

eval (Just n) = ... n ...

-- compiler checks that
-- all cases are handled
```

```kotlin
// kotlin
if (e is Nothing) {
    ....
}
if (e is Just<T>) {
    .... e.value ....
}
assert(false, "unreachable")
```

Арифметика с переменными

```
data Expr = Const Int
          | Mul Expr Expr
          | Add Expr Expr
          | Var String      -- extra construct for variables
          | Neg Expr        -- negation of expression (unary minus)
```

Арифметика с переменными

```
data Expr = Const Int
          | Mul Expr Expr
          | Add Expr Expr
          | Var String      -- extra construct for variables
          | Neg Expr        -- negation of expression (unary minus)

lookup :: Eq a => a -> [(a, b)] -> Maybe b  -- from stdlib

eval   :: [(String,Int)] -> Expr -> Maybe Int
```

Арифметика с переменными

```
data Expr = Const Int
          | Mul Expr Expr
          | Add Expr Expr
          | Var String      -- extra construct for variables
          | Neg Expr        -- negation of expression (unary minus)

lookup :: Eq a => a -> [(a, b)] -> Maybe b  -- from stdlib

eval   :: [(String,Int)] -> Expr -> Maybe Int
eval   _ (Const n) = Just n
eval env (Var s)   = lookup s env
eval env (Add l r) =    -- will be discussed later
eval env (Mul l r) =    -- will be discussed later
eval env (Neg e)   =    -- will be discussed later
```

Как вычислять недоопределенные выражения переносится на пару №2

Конец

Дальше только “запасные слайды” (их надо будет удалить) 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

N.B. Это свойство *функции*, а не языка программированая

Parametricity theorem [2]

Теорема

О параметричности. Чистые функции с параметрическим полиморфизмом работают одинаково для всех возможных типов.[a]

[a] *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

Ссылки I

OCaml for the Masses
Yaron Minsky
ссылка

Theorems for free!
Philip Wadler
ссылка

GADT tutorial
Haskell wiki