Функциональное программирование с зависимыми типами на языке Idris

Лекция 3. Типы данных и ввод-вывод

В. Н. Брагилевский

23 ноября 2017 г.

Факультет компьютерных наук, НИУ «Высшая школа экономики»

Институт математики, механики и компьютерных наук
имени И. И. Воровича, Южный федеральный университет (Ростов-на-Дону)

Интерактивная разработка

Пример: сортировка вектора (ins-sort.idr)

```
import Data.Vect

insert : Ord a => (x : a) ->
                  (xsSorted : Vect len a) ->
                  Vect (S len) a
insert x [] = [x]
insert x (y :: xs) =
   if x < y then x :: y :: xs
   else (y :: insert x xs)

inssort : Ord a => Vect n a -> Vect n a
inssort [] = []
inssort (x :: xs) = let xsSorted = inssort xs
                    in insert x xsSorted
```

Явные и неявные параметры функций (vhead.idr)

```
vhead : Vect (1 + n) elem -> elem
vhead (x :: _) = x

vhead' : (n : Nat) -> (elem : Type)
         -> Vect (1 + n) elem -> elem
vhead' n elem (x :: _) = x
```

```
idris> vhead [1,2,3]
1 : Nat
idris> vhead' 2 Nat [1,2,3]
1 : Nat
```

Использование неявных параметров (zeroes.idr)

```
zeroes : Vect n Nat
zeroes {n = Z} = []
zeroes {n = (S k)} = 0 :: zeroes

idris> zeroes
(input):Can't infer argument n to zeroes
idris> zeroes {n=5}
[0, 0, 0, 0, 0] : Vect 5 Nat

v : Vect 10 Nat
v = zeroes
```

Типы данных

Разновидности типов данных в Idris

Идеологически

- Перечислимые типы (enumerations)
- Типы-объединения (union types)
- Рекурсивные типы (recursive types)
- Обобщённые типы (generic types)
- Зависимые типы (dependent types)

Технологически (по способу объявления)

- Примитивные: Int, Double, Char, String,…
- Суммы
- Произведения

Единый синтаксис для определения типов данных

```
data typename : type  where
    alt1 : arg1 -> arg2 -> ... -> typename args
    ...
    altn : arg1 -> arg2 -> ... -> typename args
```

- `type` — `Type` или тип функции со значением типа `Type`
- `typename` — конструктор типа
- `alt*` — конструкторы значений
- если `type` это тип функции, то `typename` может принимать аргументы
- дополнительная информация может храниться либо в конструкторе типа, либо в конструкторе значения (`arg*`)

Тип-сумма: Bool (Prelude)

```
data Bool : Type where
  False : Bool
  True : Bool

b : Bool
b = True
```

Упрощённый синтаксис

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

Типы-суммы с несколькими альтернативами

Ordering (Prelude)

```
data Ordering = LT | EQ | GT

compare : Ord a => a -> a -> Ordering
```

Масти игральных карт

```
data Suit = Spades | Clubs | Diamonds | Hearts

isRed : Suit -> Bool
isRed Diamonds = True
isRed Hearts = True
isRed _ = False
```

Тип-произведение: пара символов

```
data CPair : Type where
  MkCPair : Char -> Char -> CPair

p : CPair
p = MkCPair 2 3
```

Упрощённый синтаксис

```
data CPair = MkCPair Char Char
```

Типы-произведения, параметризованные типами

Пара значений (Prelude)

```
data Pair : Type -> Type -> Type where
  MkPair : a -> b -> Pair a b
```

Пара значений в упрощённом синтаксисе

```
data Pair a b = MkPair a b
```

Пары и синтаксический сахар для них

```
p1 : Pair Bool Char
p1 = MkPair True 'x'

p2 : (Bool, Char)
p2 = (True, 'x')
```

Натуральные числа

Nat (Prelude)

```
data Nat : Type where
  Z : Nat
  S : Nat -> Nat
```

Примеры значений

```
n0 : Nat
n0 = Z
n2 : Nat
n2 = S (S Z)
n5 : Nat
n5 = S(S(S(S(S Z))))
```

Упрощённый синтаксис

```
data Nat = Z | S Nat
```

Примеры функций на натуральных числах (nats.idr)

```
prev : Nat -> Nat
prev Z = Z
prev (S x) = x

plus : Nat -> Nat -> Nat
plus Z j = j
plus (S x) j = S (plus x j)
```

Взаимно рекурсивные определения функций

```
mutual
  odd : Nat -> Bool
  odd Z = False
  odd (S k) = even k

  even : Nat -> Bool
  even Z = True
  even (S k) = odd k
```

Важные примеры параметризованных типов из Prelude

```
data Maybe a = Nothing | Just a

data Either a b = Left a | Right b

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

Список и вектор

```
data List : Type -> Type where
  Nil : List a
  (::) : (x : a) -> (xs : List a) -> List a

data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : (x : a) -> (xs : Vect k a) -> Vect (S k) a
```

Пример: циклический сдвиг вектора (rotate.idr)

```
rotate : Vect n a -> Vect n a
rotate [] = []
rotate (x :: xs) = xs ++ [x]
```

Сообщение об ошибке

```
When checking right hand side of rotate
with expected type
                Vect (S len) a
   Type mismatch between
                Vect (len + 1) a (Type of xs ++ [x])
   and
                Vect (S len) a (Expected type)
   Specifically:
                Type mismatch between
                        plus len 1
                and
                        S len
```

WTF?

```
plus len 1 /= S len
```

Источник проблемы

Определение сложения

```
plus : Nat -> Nat -> Nat
plus Z j = j
plus (S x) j = S (plus x j)
```

- Функция `plus` определена рекурсивно по первому аргументу:

  ```
  plus 1 len = plus (S Z) len
             = S (plus Z len) = S len
  ```

- Но

  ```
  plus len 1 = ???
  ```

- Алгоритму проверки типов не хватает информации о `len`.

Как бороться с такого рода проблемами?

- Метод 1: убедить алгоритм проверки типов (написать доказательство) — позже!
- Метод 2: переписать реализацию — сейчас!

Исправленная версия

```
rotate : Vect n a -> Vect n a
rotate [] = []
rotate (x :: xs) = ins_last x xs
  where
    ins_last : a -> Vect k a -> Vect (S k) a
    ins_last x [] = [x]
    ins_last x (y :: xs) = y :: ins_last x xs
```

- Явно указан тип для вложенной функции `ins_last`!

Зависимый тип: конечное множество

```
data Fin : Nat -> Type where
    FZ : Fin (S k)
    FS : Fin k -> Fin (S k)
```

Пример: значения типа Fin 3

- FZ
- FS FZ — FZ здесь имеет тип Fin 2
- FS (FS FZ) — FZ здесь имеет тип Fin 1

- Тип Fin 0 не населён
- Тип Fin n обычно используется для индексации векторов

Использование конечных множеств для индексации

```
index : Fin n -> Vect n a -> a

v : Vect 5 Nat
v = [1,2,3,4,5]

v3 : Nat
v3 = index 3 v
```

Не проходит проверку типов:

```
v6 : Nat
v6 = index 6 v
```

- Числовые литералы преобразуются к Fin n автоматически

Использование Fin и Maybe для индексирования

Преобразование Integer к Fin n

```
integerToFin :
     Integer -> (n : Nat) -> Maybe (Fin n)
```

Задача

```
tryIndex : Integer -> Vect n a -> Maybe a
```

tryIndex.idr

```
tryIndex : Integer -> Vect n a -> Maybe a
tryIndex {n} i xs =
        case integerToFin i n of
             Nothing => Nothing
             Just i' => Just (index i' xs)
```

Синтаксис записей

Определение записи

```
record Person where
     constructor MkPerson
     firstName, middleName, lastName : String
     age : Int

fred : Person
fred = MkPerson "Fred" "Joe" "Bloggs" 30
```

Доступ к полям записи

```
idris> firstName fred
"Fred" : String
idris> age fred
30 : Int
```

Обновление записи

```
idris> fred
MkPerson "Fred" "Joe" "Bloggs" 30
idris> record { firstName = "Jim" } fred
MkPerson "Jim" "Joe" "Bloggs" 30 : Person
idris> record { firstName = "Jim", age = 20 } fred
MkPerson "Jim" "Joe" "Bloggs" 20 : Person
```

Параметризованные записи

```
record Prod a b where
  constructor Times
    fst : a
    snd : b
```

Зависимые записи

```
record SizedClass (size : Nat) where
     constructor SizedClassInfo
     students : Vect size Person
     className : String


addStudent : Person -> SizedClass n
                    -> SizedClass (S n)
addStudent p c = SizedClassInfo
                     (p :: students c)
                     (className c)
```

Ввод-вывод

Функции repl и replWith

```
repl : String -> (String -> String) -> IO ()

replWith : a -> String ->
           (a -> String -> Maybe (String, a))
           -> IO ()
```

Простейший ввод-вывод

```
main : IO ()
main = putStrLn "Привет, мир"

main : IO ()
main = do putStrLn "Введите ваше имя: "
          name <- getLine
          putStrLn ("Привет, " ++ name ++ "!")
```

Пример: ввод натурального числа

```
Data.String.parsePositive : Num a =>
                            String -> Maybe a
```

```
import Data.String

parseNat : String -> Maybe Nat
parseNat = parsePositive

readNat : IO (Maybe Nat)
readNat = do
    s <- getLine
    case parseNat s of
      Nothing => pure Nothing
      Just x => pure (Just x)
```

Пример: ввод вектора

```
readVectLen : (len : Nat) ->
              IO (Vect len String)
readVectLen Z = pure []
readVectLen (S k) = do x <- getLine
                       xs <- readVectLen k
                       pure (x :: xs)
```

Проблема: а если длина заранее неизвестна?

```
readVect : IO (Vect len String)
```

```
readVect : {len : Nat} -> IO (Vect len String)
```

- Совпадает по смыслу с предыдущей версией.

Чтение вектора неизвестной длины (readVect.idr)

```
readVect : IO (len ** Vect len String)
readVect = do s <- getLine
              if s == ""
                 then pure (_ ** [])
                 else do (_ ** ss) <- readVect
                         pure (_ ** s :: ss)
```

- Зависимая пара

Использование зависимых пар: filter

```
filter : (a -> Bool) -> Vect n a -> ???

filter : (a -> Bool) -> Vect n a
                     -> (p ** Vect p a)

filter f [] = (_ ** [])
filter f (x :: xs) =
  let (_ ** xs') = filter f xs in
  if f x then (_ ** x :: xs')
         else (_ ** xs')
```

Пример: спаривание строк двух векторов

- Считываем два вектора.
- Если они разной длины, то ошибка.
- Помещаем соответствующие элементы векторов одинаковой длины в пары.

```
Data.Vect.zip : Vect n a -> Vect n b ->
                Vect n (a, b)
```

Пример: спаривание строк двух векторов

```
zipInputs : IO ()
zipInputs = do
    (l1 ** v1) <- readVect
    (l2 ** v2) <- readVect
    case exactLength l1 v2 of
        Nothing => putStrLn "Error"
        Just v => printLn (zip v1 v)
```

Список литературы

Brady, Edwin (March, 2017). *Type-Driven Development with Idris*. Manning Publications.