Функциональное программирование с зависимыми типами на языке Idris

Лекция 8. Представления и тотальность

В. Н. Брагилевский

29 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)

Идея представлений (views)

Представления списка

Список: голова и хвост

```
describeList : List Int -> String
describeList [] = "Empty"
describeList (x :: xs) =
        "Non-empty, tail = " ++ show xs
```

Список: начало + последний элемент (не работает!)

```
describeListEnd : List Int -> String
describeListEnd [] = "Empty"
describeListEnd (xs ++ [x]) =
        "Non-empty, initial part = " ++ show xs
```

Тип данных для нового представления (View)

```
data ListLast : List a -> Type where
     Empty : ListLast []
     NonEmpty : (xs : List a) -> (x : a) ->
                ListLast (xs ++ [x])
```

Покрывающая функция (covering function)

```
listLast : (xs : List a) -> ListLast xs
listLast [] = Empty
listLast (x :: xs) =
   case listLast xs of
     Empty => NonEmpty [] x
     NonEmpty xs y => NonEmpty (x :: xs) y
```

Использование представления

```
describeListEnd : List Int -> String
```

```
describeHelper : (input : List Int) ->
                 ListLast input -> String
describeHelper [] Empty = "Empty"
describeHelper (xs ++ [x]) (NonEmpty xs x)
    = "Non-empty, initial part = " ++ show xs

describeListEnd : List Int -> String
describeListEnd xs =
        describeHelper xs (listLast xs)
```

Конструкция with

```
describeListEnd : List Int -> String
describeListEnd xs with (listLast xs)
 describeListEnd [] | Empty = "Empty"
 describeListEnd (ys ++ [x]) | (NonEmpty ys x)
   = "Non-empty, initial part = " ++ show ys
```

Реализация `parity` с помощью `with`

```
data Parity : Nat -> Type where
    Even : Parity (n + n)
    Odd  : Parity (S (n + n))

parity : (n : Nat) -> Parity n
parity Z = Even {n = Z}
parity (S Z) = Odd {n = Z}
parity (S (S k)) with (parity k)
  parity (S (S (n + n))) | Even =
      rewrite plusSuccRightSucc n n
      in Even {n = S n}
  parity (S (S (S (n + n)))) | Odd =
      rewrite plusSuccRightSucc n n
      in Odd {n = S n}
```

Обращение списка с использованием представления

```
myReverse : List a -> List a
myReverse input with (listLast input)
  myReverse [] | Empty = []
  myReverse (xs ++ [x]) | (NonEmpty xs x) =
                          x :: myReverse xs
```

Две проблемы

1. Это обращение крайне неэффективно.
2. Функция myReverse не распознаётся как тотальная:

```
idris> :total myReverse
Main.myReverse is possibly not total due to:
    with block in Main.myReverse, which is possibly
      not total due to recursive path:
    with block in Main.myReverse, Main.myReverse,
      Main.myReverse
```

Контроль тотальности (завершаемости)

- Команда интерпретатора `:total имя_функции`
- Директива `%default total` в коде программы
- Спецификатор `total` перед типовой аннотацией функции
- Консервативный характер проверок
- Общая идея проверки: аргумент рекурсивного вызова должен быть структурно «меньше»
- Ошибки в реализации контроля тотальности

Рекурсивные представления

Представление

```
data SnocList : List a -> Type where
    Empty : SnocList []
    Snoc : (rec : SnocList xs) ->
           SnocList (xs ++ [x])
```

Покрывающая функция

```
snocListHelp : (snoc : SnocList input) ->
               (rest : List a) ->
               SnocList (input ++ rest)
snocListHelp {input} snoc []
     = rewrite appendNilRightNeutral input
       in snoc
snocListHelp {input} snoc (x :: xs)
     = rewrite appendAssociative input [x] xs
       in snocListHelp (Snoc snoc) xs

snocList : (xs : List a) -> SnocList xs
snocList xs = snocListHelp Empty xs
```

Обращение списка

```
myReverse_help : (input : List a) ->
                 SnocList input ->
                 List a
myReverse_help [] Empty = []
myReverse_help (xs ++ [x]) (Snoc rec) =
               x :: myReverse_help xs rec

myReverse1 : List a -> List a
myReverse1 input =
   myReverse_help input (snocList input)
```

Обращение списка: рекурсивный `with`

```
myReverse : List a -> List a
myReverse input with (snocList input)
  myReverse [] | Empty = []
  myReverse (xs ++ [x]) | (Snoc rec) =
                    x :: myReverse xs | rec
```

Представления списков в стандартной библиотеке

- VList — первый элемент, середина списка, последний элемент
- Split — деление списка пополам, нерекурсивно, с выделением голов половин
- SplitRec — деление списка пополам, рекурсивно
- SnocList — обход с конца, рекурсивно
- Filtered — обход элементов, удовлетворяющих условию, рекурсивно

Представление `SplitRec` (Data.List.Views)

```
data SplitRec : List a -> Type where
  SplitRecNil : SplitRec []
  SplitRecOne : SplitRec [x]
  SplitRecPair : (lrec : Lazy (SplitRec ls))
                 -> (rrec : Lazy (SplitRec rs))
                 -> SplitRec (ls ++ rs)
```

Сортировка слиянием

```
mergeSort : Ord a => List a -> List a
mergeSort input with (splitRec input)
  mergeSort [] | SplitRecNil = []
  mergeSort [x] | SplitRecOne = [x]
  mergeSort (lefts ++ rights) |
                        (SplitRecPair lrec rrec)
               = merge (mergeSort lefts | lrec)
                       (mergeSort rights | rrec)
```

Представления для `Nat` (Data.Nat.Views)

- Half
- HalfRec

Тотальная реализация верифицированного двоичного представления

```
import Data.Nat.Views

%default total

data Binary : Nat -> Type where
    BEnd : Binary Z
    BO : Binary n -> Binary (n + n)
    BI : Binary n -> Binary (S (n + n))
```

```
natToBinV : (n:Nat) -> Binary n
natToBinV n with (halfRec n)
  natToBinV Z | HalfRecZ = BEnd
  natToBinV (x + x) | (HalfRecEven rec)
                    = BO (natToBinV x | rec)
  natToBinV (S (x + x)) | (HalfRecOdd rec)
                    = BI (natToBinV x | rec)
```

Пример 1

```
main : IO ()
main = do
    CALL g ON 10 WITH 5
    CALL g ON 1 WITH 3
    CALL h ON "QQ" WITH False
```

Пример 2

```
main : IO ()
main = do for x in [1..10]:
                do putStr ("Number " ++ show x)
                   putStrLn ""
           putStrLn "Done!"
```

Определение EDSL в Idris

Определения

- DSL — domain-specific language — язык, специализированный для использования в конкретной предметной области (*в отличие от языков общего назначения*).
- EDSL — embedded DSL — язык, реализованный в виде библиотеки, использующей синтаксис *базового языка* или его подмножество и одновременно добавляющей сущности предметной области (типы данных, функции и пр.) — фрагменты *целевого языка*.
- Определение EDSL необязательно означает расширение синтаксиса базового язык, к тому же иногда предполагается его упрощение.

Поддержка определения EDSL в Idris

- Реализация сущностей предметной области в системе типов Idris
- Расширение do-нотации
- Определение новых синтаксических правил
- Перегрузка синтаксиса базового языка для использования в целевом языке (крайне ограничено)

Определение EDSL в Idris

Расширение do-нотации

Странные вещи в do-блоках

```
sum : Int
sum = do
        15
        15
        -5
        19
        -2

(>>=) : Int -> (Int -> Int) -> Int
(>>=) n f = n + f 0

Idris> sum
42 : Int
```

```
(>>=) : String -> (String -> String) -> String
(>>=) n f = n ++ f ""

sum : String
sum = do "15"
         "10"
```

```
(>>=) : String -> (String -> List String)
        -> List String
(>>=) n f = n :: f ""
syntax END = []

sum2 : List String
sum2 = do "10"
          "20"
          "30"
          END
```

Определение EDSL в Idris

Введение синтаксических правил

```
syntax CALL [f] ON [t] WITH [a] = f t a;

g : Int -> Int -> IO ()
g a b = printLn (a + b)

h : String -> Bool -> IO ()
h s False = printLn s
h s True = printLn ""

main : IO ()
main = do
    CALL g ON 10 WITH 5
    CALL g ON 1 WITH 3
    CALL h ON "QQ" WITH False
```

- Преобразование выражений в вызовы функций

Другие примеры

```
syntax [var] ":=" [val] = Assign var val
syntax [test] "?" [t] ":" [e]
                    = if test then t else e
syntax select [x] from [t] "where" [w]
                         = SelectWhere x t w
syntax select [x] from [t] = Select x t
```

- Ключевые слова и символы пишутся в кавычках

```
syntax for {x} "in" [xs] ":" [body]
                        = for xs (\x => body)

for : (Traversable t, Applicative f) =>
      t a -> (a -> f b) -> f (t b)

main : IO ()
main = do for x in [1..10]:
              putStrLn ("Number " ++ show x)
          putStrLn "Done!"

main : IO ()
main = do for x in [1..10]:
              do putStr ("Number " ++ show x)
                 putStrLn ""
          putStrLn "Done!"
```

- Связанные переменные пишутся в {}.

НЕ ВЕРИТЕ?

Список литературы


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


The Idris Tutorial. URL: http://docs.idris-lang.org/en/latest/tutorial/index.html.


Theorem Proving. URL: http://docs.idris-lang.org/en/latest/proofs/index.html.