Программирование с зависимыми типами на языке Idris

Лекция 4. Типы как сущности первого класса, функции на типах

В. Н. Брагилевский

12 февраля 2017 г.

Computer Science клуб (Санкт-Петербург)

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

Синонимы типов

```
import Data.Vect

tri : Vect 3 (Double, Double)
tri = [(0.0, 0.0), (3.0, 0.0), (0.0, 4.0)]

Position : Type
Position = (Double, Double)

tri' : Vect 3 Position
tri' = [(0.0, 0.0), (3.0, 0.0), (0.0, 4.0)]

Polygon : Nat -> Type
Polygon n = Vect n Position

tri'' : Polygon 3
tri'' = [(0.0, 0.0), (3.0, 0.0), (0.0, 4.0)]
```

Функции на типах

Функции the и cast

```
the : (a : Type) -> a -> a
cast : Cast from to => from -> to


summate : Int -> String -> Maybe (String, Int)
summate s y = case the Int (cast y) of
                   0 => Nothing
                   n => let s' = s + n in
                        Just ("Sum=" ++ cast s' ++ "\n", s')

main : IO ()
main = replWith 0 "> " summate
```

Сопоставление с образцом для типов

```
data Ty = TyNat | TyBool | TyString

evalType : Ty -> Type

initVal : (ty : Ty) -> evalType ty

toString : (ty : Ty) -> evalType ty -> String
```

TypeFuns.idr

```
data Ty = TyNat | TyBool | TyString

evalType : Ty -> Type
evalType TyNat = Nat
evalType TyBool = Bool
evalType TyString = String

initVal : (ty : Ty) -> evalType ty
initVal TyNat = 0
initVal TyBool = False
initVal TyString = ""

toString : (ty : Ty) -> evalType ty -> String
toString TyNat x = "Nat: " ++ show x
toString TyBool x = "Bool: " ++ show x
toString TyString x = "String: " ++ trim x
```

Функция evalType не нужна!

```
data Ty = TyNat | TyBool | TyString

toString : (ty : Ty) -> ?evalType -> String
```

TypeFuns.idr

```
toString : (ty : Ty) -> (case ty of
                              TyNat => Nat
                              TyBool => Bool
                              TyString => String) -> String
```

Функции с переменным числом аргументов

```
adder  0  10       = 10
adder  1  0  5     = 5
adder  2  0  4  6  = 20
```

Аргументы функции adder:

- количество дополнительных аргументов
- начальное значение
- дополнительный аргумент
- ...

- Тип функции adder можно вычислять:

```
adder 0 : Int -> Int
adder 1 : Int -> Int -> Int
adder 2 : Int -> Int -> Int -> Int
```

Идея решения

- Функция AdderType, вычисляющая тип функции по заданному числу дополнительных аргументов.
- Собственно функция adder этого типа.

adder.idr

```
AdderType : (numargs : Nat) -> Type
AdderType Z = Int
AdderType (S k) = Int -> AdderType k

adder : (numargs : Nat) -> (acc : Int) -> AdderType numargs
adder Z acc = acc
adder (S k) acc = \i => adder k (acc+i)
```

Обобщённая версия

```
AdderType : (numargs : Nat) -> (t : Type) -> Type
AdderType Z t = t
AdderType (S k) t = t -> AdderType k t
adder : Num t => (numargs : Nat) -> (acc : t)
                -> AdderType numargs t
adder Z acc = acc
adder (S k) acc = \i => adder k (acc+i)
```

Типобезопасная функция sprintf (1)

Использование

```
sprintf "Hello!"                    = "Hello!"
sprintf "Answer : %d"  2            = "Answer : 42"
sprintf "%s number %d" "Slide" 8 = "Slide number 8"
```

Компоненты

- Тип данных для описания форматов
- Функция из строк в тип данных для форматов
- Функция на типах, вычисляющая тип `sprintf`

Типобезопасная функция sprintf (2)

Типы sprintf

```
sprintf "Hello!"        : String
sprintf "Answer: %d"    : Int -> String
sprintf "%s number %d" : String -> Int -> String
```

Тип данных для форматов

```
data Format = Number Format
            | Str Format
            | Lit String Format
            | End
```

"%s = %d" $\Longrightarrow$ `Str (Lit " = " (Number End))`

Типобезопасная функция sprintf (3)

```
SPrintfType : Format -> Type

sprintfFmt : (fmt : Format) ->
             (acc : String) ->
             SPrintfType fmt

toFormat : (xs : List Char) -> Format

sprintf : (fmt : String) ->
          SPrintfType (toFormat (unpack fmt))
```

sprintf.idr

```
data Format = Number Format
            | Str Format
            | Lit String Format
            | End

%name Format fmt

SPrintfType : Format -> Type
SPrintfType (Number fmt) = Int -> SPrintfType fmt
SPrintfType (Str fmt) = String -> SPrintfType fmt
SPrintfType (Lit x fmt) = SPrintfType fmt
SPrintfType End = String
```

```
sprintfFmt : (fmt : Format) -> (acc : String)
                            -> SPrintfType fmt
sprintfFmt (Number fmt) acc =
                \k => sprintfFmt fmt (acc ++ show k)
sprintfFmt (Str fmt) acc = \s => sprintfFmt fmt (acc ++ s)
sprintfFmt (Lit x fmt) acc = sprintfFmt fmt (acc ++ x)
sprintfFmt End acc = acc
```

```
toFormat : (xs : List Char) -> Format
toFormat [] = End
toFormat ('%' :: 'd' :: xs) = Number (toFormat xs)
toFormat ('%' :: 's' :: xs) = Str (toFormat xs)
toFormat ('%' :: xs) = Lit "%" (toFormat xs)
toFormat (x :: xs)  = case toFormat xs of
                        (Lit s fmt) => Lit (strCons x s) fmt
                        fmt => Lit (singleton x) fmt
```

```
sprintf : (fmt : String)
          -> SPrintfType (toFormat (unpack fmt))
sprintf fmt = sprintfFmt _  ""

main : IO ()
main = do
  s <- getLine
  print $ sprintf "%s = %d" s 2
```

Список литературы

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