Программирование с зависимыми типами на языке Idris

Лекции 9–10. Разное

В. Н. Брагилевский

19 февраля 2017 г.

Computer Science клуб (Санкт-Петербург)

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

Содержание

Верификация протоколов на типах

Академический процесс

- Написание статей
- Подача статьи на рецензию и рецензирование с принятием или отклонением
- Нельзя подавать одну и ту же работу дважды или отклонять не поданную
- Реализовывать этот процесс мы конечно же не будем (мы тут не ИИ изучаем в конце концов!)
- Мы опишем прототип процесса на типах.

PaperState и Paper

```
data PaperState = Written | Reviewing | Accepted | Rejected

data Paper : PaperState -> Type where
  MkPaper : Paper s
```

События академического процесса

- написание
- подача
- принятие
- отклонение
- исправление

PaperEvent

```
data PaperEvent : Type -> Type where
  Write  :                      PaperEvent (Paper Written)
  Submit : Paper Written    -> PaperEvent (Paper Reviewing)
  Accept : Paper Reviewing -> PaperEvent (Paper Accepted)
  Reject : Paper Reviewing -> PaperEvent (Paper Rejected)
  Revise : Paper Rejected   -> PaperEvent (Paper Reviewing)
```

Программирование без функций

- У нас есть набор действий, которые должны выстраиваться в последовательность
- Похоже на монаду!

```
data PaperLang : Type -> Type where
  Action : PaperEvent a -> PaperLang a
  (>>=) : PaperLang a -> (a -> PaperLang b) -> PaperLang b
```

- Здесь нет функций, только конструкторы данных

Как написать статью?

```
prog1 : PaperLang (Paper Accepted)
prog1 = Action Write >>= Action . Submit
                     >>= Action . Accept

prog2 : PaperLang (Paper Accepted)
prog2 = do
   p <- Action Write
   p <- Action (Submit p)
   p <- Action (Reject p)
   p <- Action (Revise p)
   Action (Accept p)
```

papers.idr: давайте почитерим!

Что такое prog2?

```
Idris> :printdef prog2
prog2 : PaperLang (Paper Accepted)
prog2 = ((Action Write) >>=
         (\p =>
            ((Action (Submit p)) >>=
             (\p5 =>
                ((Action (Reject p5)) >>=
                 (\p8 => ((Action (Revise p8)) >>=
                     (\p11 => Action (Accept p11)))))))))
```

- Это значение типа данных, построенное конструктором данных (>>=).

Небольшое улучшение: неявные функции

```
implicit
action : PaperEvent a -> PaperLang a
action = Action

prog2' : PaperLang (Paper Accepted)
prog2' = do
   p <- Write
   p <- Submit p
   p <- Reject p
   p <- Revise p
   Accept p
```

- Неявные функции вызываются автоматически для удовлетворения алгоритма проверки типов

Добавление новых ключевых слов

```
syntax write = Action (Write)
syntax submit = \p => Action (Submit p)
syntax accept = \p => Action (Accept p)
syntax reject = \p => Action (Reject p)
syntax revise = \p => Action (Revise p)
syntax AcceptedPaper = PaperLang (Paper Accepted)

prog3 : AcceptedPaper
prog3 = write >>= submit >>= accept

prog4 : AcceptedPaper
prog4 = write >>= submit >>= reject >>= revise >>=
                             reject >>= revise >>= accept
```

Определение EDSL

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

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

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?

Определение EDSL

Пример: интерпретатор выражений с типами

Выражения и типы

```
e ::=
    n                       (число)
    x                       (переменная)
    λx.e                    (лямбда)
    e e                     (применение)
    e ○ e                   (операция)
    if e then e else e      (условие)

t ::=
    int
    bool
    t → t
```

Окружение и контекст типизации

- Окружение содержит значения переменных
- Контекст типизации содержит типы переменных

$$\Gamma ::= \varnothing \mid \Gamma, x : T$$

Правила типизации

$$\Gamma \vdash e : T$$

$$\frac{n - \text{число}}{\varnothing \vdash n : int} \quad (Val) \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (Var)$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \to T_2} \quad (Lam)$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 e_2 : T_2} \quad (App)$$

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : T} \quad (If)$$

$$\frac{\circ - \text{операция над } T_1 \text{ и } T_2 \text{ с рез. в } T_3 \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \circ e_2 : T_3} \quad (Op)$$

Индексы де Брауна (De Bruijn)

$$\lambda x.\lambda y.x(yx) \quad \Longrightarrow \quad \lambda.\lambda.1'(0'1')$$

$$x + y \quad \Longrightarrow \quad 0' + 1'$$

где $\Gamma = [\ldots, int, int]$

- Индексы де Брауна напрямую соответствуют (обращённой) позиции в контексте типизации и окружении

Типы и их интерпретация

```
data Ty = TyInt | TyBool | TyFun Ty Ty

interpTy : Ty -> Type
interpTy TyInt       = Int
interpTy TyBool      = Bool
interpTy (TyFun s t) = interpTy s -> interpTy t
```

Окружение и контекст типизации

```
using (G : Vect n Ty)

  data Env : Vect n Ty -> Type where
      Nil  : Env Nil
      (::) : interpTy a -> Env G -> Env (a :: G)

  data HasType : (i : Fin n) -> Vect n Ty -> Ty -> Type
                                                  where
      Stop : HasType FZ (t :: G) t
      Pop  : HasType k G t -> HasType (FS k) (u :: G) t
```

- `HasType i G t` означает, что $\Gamma \vdash i' : t$, где i' — индекс де Брауна

Поиск значения

```
lookup : HasType i G t -> Env G -> interpTy t
lookup Stop    (x :: xs) = x
lookup (Pop k) (x :: xs) = lookup k xs
lookup Stop    [] impossible
```

Выражения

```
data Expr : Vect n Ty -> Ty -> Type where
  Var : HasType i G t -> Expr G t
  Val : (x : Int) -> Expr G TyInt
  Lam : Expr (a :: G) t -> Expr G (TyFun a t)
  App : Lazy (Expr G (TyFun a t)) -> Expr G a -> Expr G t
  Op  : (interpTy a -> interpTy b -> interpTy c) ->
        Expr G a -> Expr G b -> Expr G c
  If  : Expr G TyBool -> Expr G a -> Expr G a -> Expr G a
```

Интерпретация выражений

```
total
interp : Env G -> (e : Expr G t) -> interpTy t
interp env (Var i)     = lookup i env
interp env (Val x)     = x
interp env (Lam sc)    = \x => interp (x :: env) sc
interp env (App f s)   = (interp env f) (interp env s)
interp env (Op op x y) = op (interp env x) (interp env y)
interp env (If x t e)  = if interp env x
                           then interp env t
                           else interp env e
```

Проверка

$$\lambda x.\lambda y.y + x \quad \Longrightarrow \quad \lambda.\lambda.0' + 1'$$

```
func : Expr G (TyFun TyInt (TyFun TyInt TyInt))
func = Lam (Lam (Op (+) (Var Stop) (Var (Pop Stop))))

e : Expr G TyInt
e = App (App func (Val 5)) (Val 10)

Idris> interp [] e
15:int
```

Перегрузка синтаксиса

```
lam_ : TTName -> Expr (a :: G) t -> Expr G (TyFun a t)
lam_ _ = Lam

dsl expr
    lambda = lam_
    variable = Var
    index_first = Stop
    index_next = Pop
```

```
eId : Expr G (TyFun TyInt TyInt)
eId = expr (\x => x)

eAdd : Expr G (TyFun TyInt (TyFun TyInt TyInt))
eAdd = expr (\x, y => Op (+) x y)

eDouble : Expr G (TyFun TyInt TyInt)
eDouble = expr (\x => App (App eAdd x) (Var Stop))
```

```
  eFac : Expr G (TyFun TyInt TyInt)
  eFac = expr (\x => If (Op (==) x (Val 0))
                        (Val 1)
                        (Op (*) (App eFac (Op (-) x (Val 1))) x))

testFac : Int
testFac = interp [] eFac 4

main : IO ()
main = printLn testFac
```

Представления и конструкция with

Пример: доказательная чётность-нечётность

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

parity : (n : Nat) -> Parity n
```

```
parity_lemma_1 : (j : Nat) -> Parity ((S j) + (S j))
                           -> Parity (S (S (plus j j)))
parity_lemma_1 j par = rewrite plusSuccRightSucc j j in par

parity_lemma_2 : (j : Nat) -> Parity (S ((S j) + (S j)))
                           -> Parity (S (S (S (plus j j))))
parity_lemma_2 j par = rewrite plusSuccRightSucc j j in par

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 (j + j))) | Even =
                         parity_lemma_1 j (Even {n=S j})
  parity (S (S (S (j + j)))) | Odd =
                         parity_lemma_2 j (Odd {n=S j})
```

Преобразование натурального числа в двоичную форму

```
data Digit = I | O

natToBin : Nat -> List Digit

natToBin Z = []
natToBin k with (parity k)
  natToBin (n + n) | Even = O :: natToBin n
  natToBin (S (n + n)) | Odd = I :: natToBin n
```

Верифицированное двоичное представление

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

natToBin : (n:Nat) -> Binary n

natToBin Z = BEnd
natToBin k with (parity k)
  natToBin (n + n) | Even = BO (natToBin n)
  natToBin (S (n + n)) | Odd = BI (natToBin n)
```

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 portion = " ++ 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
describeListEnd xs with (listLast xs)
  describeListEnd [] | Empty = "Empty"
  describeListEnd (ys ++ [x]) | (NonEmpty ys x) =
              "Non-empty, initial portion = " ++ show ys
```

Реализация простых игр: «Виселица»

hangman.idr (код из Brady, Март, 2017)

Что дальше

1. Изучайте Idris
2. Пишите библиотеки и приложения
3. Изучайте Haskell
4. Исправляйте ошибки в компиляторе
5. ???
6. PROFIT!

Список литературы


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


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