Функциональное программирование с зависимыми типами на языке Idris

Лекции 9–10. Разное

В. Н. Брагилевский

1 декабря 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)

Бесконечные данные и процессы

Потоки данных (streams)

Создание бесконечного потока

```
iterate : (a -> a) -> a -> Stream a
iterate f x = x :: iterate f (f x)

nats : Stream Nat
nats = iterate (+1) 0

take : (n : Nat) -> (xs : Stream a) -> List a
take Z _ = []
take (S n) (x :: xs) = x :: take n xs
```

```
idris> take 10 nats
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] : List Nat
```

Числа Фибоначчи

```
fibs : Stream Nat
fibs = map fst (iterate next (0,1,1))
  where
    next : (Nat, Nat, Nat) -> (Nat, Nat, Nat)
    next (_, a, b) = (a, b, a+b)
    fst : (Nat, Nat, Nat) -> Nat
    fst (a, _, _) = a

map : (a -> b) -> Stream a -> Stream b
map f (x :: xs) = f x :: map f xs
```

```
idris> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34] : List Nat
```

Числа Фибоначчи и «Завязывание узлов»

```
fibs' : Stream Nat
fibs' = 0 :: 1 :: zipWith (+) fibs'
                          (tail fibs')
```

```
idris> take 10 fibs'
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34] : List Nat
```

Отложенные вычисления

```
data DelayReason = Infinite | LazyValue

data Delayed : DelayReason -> Type -> Type
                                    where
     Delay : (val : a) -> Delayed t a

Force : Delayed t a -> a
Force (Delay x) = x
```

- `Delay` и `Force` вставляются компилятором автоматически (при соответствующем типе).

Псевдонимы типов отложенных вычислений

```
Lazy : Type -> Type
Lazy t = Delayed LazyValue t

Inf : Type -> Type
Inf t = Delayed Infinite t
```

Тип Stream

```
data Stream : Type -> Type where
  (::) : (value : elem) -> Inf (Stream elem)
                        -> Stream elem

iterate : (a -> a) -> a -> Stream a
iterate f x = x :: Delay (iterate f (f x))

take : (n : Nat) -> (xs : Stream a) -> List a
take Z _ = []
take (S n) (x :: xs) = x :: take n (Force xs)
```

```
idris> :total iterate
Prelude.Stream.iterate is Total
```

Тотальность

- *Завершаемость*
- *Продуктивность*: способность произвести непустой конечный префикс бесконечного результата за конечное время

```
iterate : (a -> a) -> a -> Stream a
iterate f x = x :: Delay (iterate f (f x))
```

Бесконечные процессы как последовательности действий

Действие

```
data InfIO : Type where
     Do : IO a -> (a -> Inf InfIO) -> InfIO

loopPrint : String -> InfIO
loopPrint msg = Do (putStrLn msg)
                   (\_ => loopPrint msg)
```

```
idris> :total loopPrint
Main.loopPrint is Total
```

- Действие должно быть выполнено.

Выполнение действий

```
run : InfIO -> IO ()
run (Do c f) =
  do
    res <- c
    run (f res)
```

```
idris> :total run
Main.run is possibly not total
due to recursive path:
    Main.run, Main.run
```

Делаем run тотальной!

Странный тип данных

```
data More = M (Lazy More)
```

Новая версия run (тотальная)

```
run : More -> InfIO -> IO ()
run (M more) (Do c f) =
    do res <- c
       run more (f res)
```

```
idris> :total run
Main.run is Total
```

Inf и Lazy

- `Inf` и `Lazy` участвуют в контроле тотальности
- `Inf` учитывается проверкой продуктивности
- `Lazy` учитывается проверкой завершаемости: считается что `more` *структурно меньше*, чем `M more`
- Во время выполнения `Inf` и `Lazy` работают одинаково: вычисление откладывается до момента фактического использования значения

Запуск бесконечного процесса

Носитель бесконечности

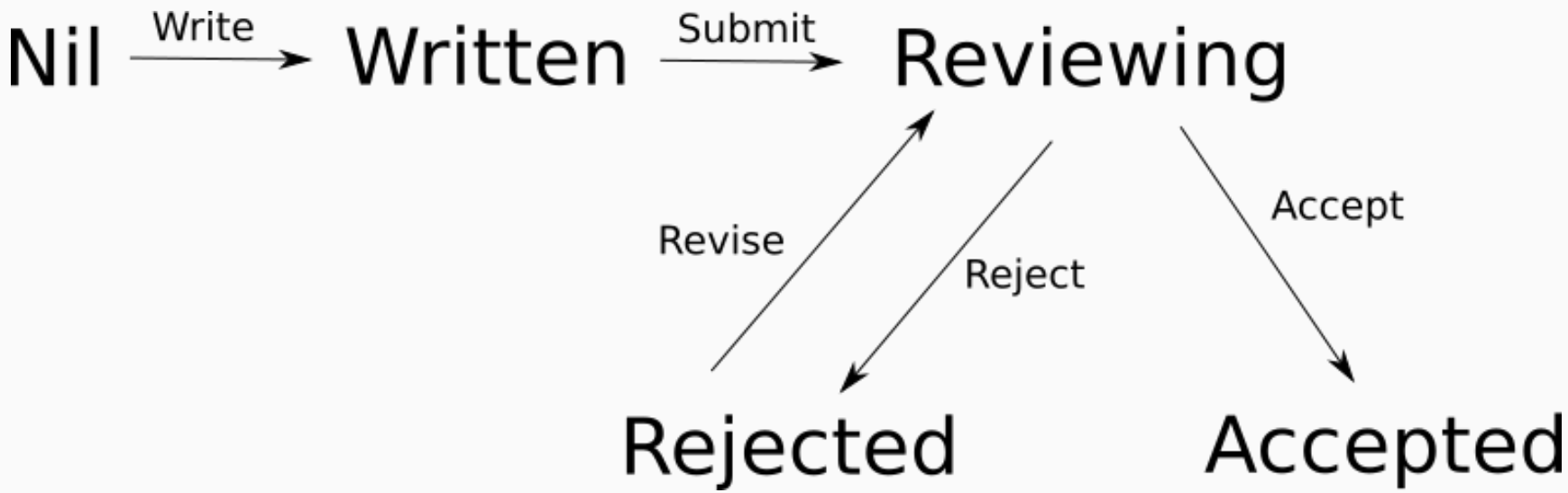
```
partial
forever : More
forever = M forever
```

Функция main

```
partial
main : IO ()
main = run forever (loopPrint "hi!")
```

Конечные автоматы и верификация протоколов

Как писать статьи



Наша цель

- Описать состояния и переходы типами так, чтобы проверка типов могла всё контролировать.
- Организовать описание корректных последовательностей переходов для описания процесса получения принятой статьи из ещё не написанной.

Состояния

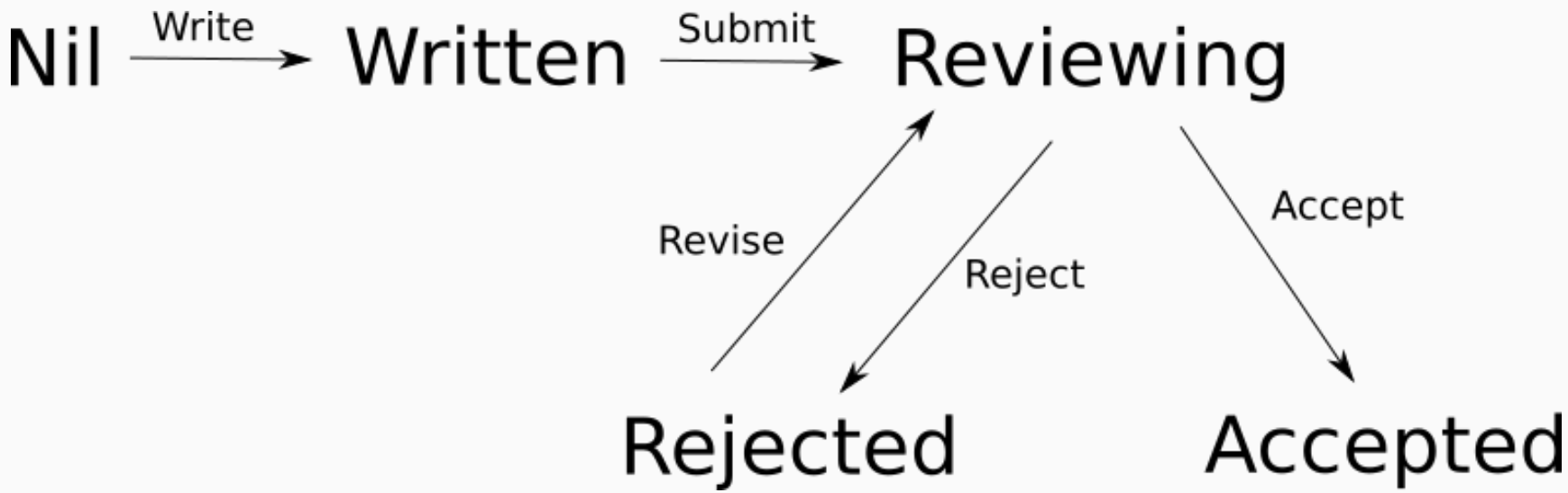
```
data PaperState = Nil | Written | Reviewing
                | Accepted | Rejected
```

Переходы и их комбинирование

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

  Pure : ty -> PaperEvent ty state state
  (>>=) : PaperEvent a state1 state2 ->
          (a -> PaperEvent b state2 state3) ->
          PaperEvent b state1 state3
```

Пишем программы для написания статей



Идеальный вариант

```
prog1 : PaperEvent () Nil Accepted
prog1 = do
    Write
    Submit
    Accept
```

Вариант чуть похуже

```
prog2 : PaperEvent () Nil Accepted
prog2 = do
    Write
    Submit
    Reject
    Revise
    Accept
```

Вендинговая машина: монеты с шоколадками

```
VendState : Type
VendState = (Nat, Nat)
```

Базовые операции

```
data MachineCmd : Type -> VendState ->
                  VendState -> Type where
  InsertCoin : MachineCmd ()
                          (pounds, chocs)
                          (S pounds, chocs)
  Vend       : MachineCmd ()
                          (S pounds, S chocs)
                          (pounds, chocs)
  GetCoins   : MachineCmd ()
                          (pounds, chocs)
                          (Z, chocs)
```

Пользовательский ввод

```
data Input = COIN
           | VEND
           | CHANGE
           | REFILL Nat
```

Остальные операции (в MachineCmd)

```
Display : String ->
          MachineCmd () state state
Refill : (bars : Nat) ->
          MachineCmd ()
                     (Z, chocs)
                     (Z, bars + chocs)

GetInput : MachineCmd (Maybe Input) st st

Pure : ty -> MachineCmd ty state state
(>>=) : MachineCmd a state1 state2 ->
        (a -> MachineCmd b state2 state3) ->
        MachineCmd b state1 state3
```

Компиляция кода на Idris

Внутренний язык TT

TT — это λ-исчисление с зависимыми типами, индуктивными семействами и определениями через сопоставления с образцом.

- Термы: константы, переменные, применение, конструкторы типов и данных.
- Константы: $Type_i$, целочисленные и строковые литералы.
- Связыватели (bindings): λ-абстракции, `let`-определения, типовые абстракции ($\forall x : t.f$).

Программа на языке TT — это коллекция индуктивных определений (данные) и определений функций.

Виды определений

Индуктивные определения

```
data T : t where D1 : t | ... | Dn : t
```

Определения на основе сопоставления с образцом

```
f : t
var x1 : t1 . f t1 = t1
...
var xn : tn . f tn = tn
```

Тип Vect в Idris

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : a ->
         Vect k a ->
         Vect (S k) a
```

Тип Vect в TT

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

Функция vadd в Idris

```
vadd : Num a => Vect n a ->
       Vect n a -> Vect n a
vadd [] [] = []
vadd (x :: xs) (y :: ys) = x + y ::
                           vadd xs ys
```

Функция vadd в TT

```
vadd : (a : Type) -> (n : Nat) -> Num a ->
       Vect n a -> Vect n a -> Vect n a

var a : Type, c : Num a.
  vadd a Z c (Nil a) (Nil a) = Nil a

var a : Type, k : Nat, c : Num a,
    x : a, xs : Vect k a, y : a, ys : Vect k a.
  vadd a (S k) c ((::) a k x xs) ((::) a k y ys)
     = ((::) a k ((+) c x y) (vadd a k c xs ys))
```

Схема трансляции

$$Idris \longrightarrow Idris^{-} \longrightarrow TT \longrightarrow Executable$$

1. Рассахаривание (desugaring)
2. Насыщение (elaboration)
3. Стирание типов и кодогенерация

Пример насыщения

```
vadd : Num a => Vect n a ->
       Vect n a -> Vect n a
vadd [] [] = []
vadd (x :: xs) (y :: ys) = x + y ::
                           vadd xs ys
```

Пример насыщения (2)

```
vadd : (a : _) ->  (n : _) ->
       Num a -> Vect n a ->
       Vect n a -> Vect n a
vadd _ _ c (Nil _) (Nil _) = (Nil _)
vadd _ _ c ((::) _ _ x xs)
           ((::) _ _ y ys)
   = (::) _ _ (+ _ x  y)
              (vadd _ _ _ xs ys)
```

Пример насыщения (3)

```
vadd : (a : Type) ->  (n : Nat) ->
       Num a -> Vect n a ->
       Vect n a -> Vect n a
vadd a 0 c (Nil a) (Nil a) = (Nil a)
vadd a (S k) c ((::) a k x xs)
               ((::) a k y ys)
    = (::) a k (+ _ x  y)
               (vadd a k c xs ys)
```

Результат насыщения (код TT)

```
vadd : (a : Type) -> (n : Nat) -> Num a ->
       Vect n a -> Vect n a -> Vect n a

var a : Type, c : Num a.
  vadd a Z c (Nil a) (Nil a) = Nil a

var a : Type, k : Nat, c : Num a,
    x : a, xs : Vect k a, y : a, ys : Vect k a.
  vadd a (S k) c ((::) a k x xs) ((::) a k y ys)
     = ((::) a k ((+) c x y) (vadd a k c xs ys))
```

Как это всё работает?

- TT_{dev} — TT с двумя дополнительными конструкциями, соответствующими недосформированным термам (holes, guesses)
- Глобальное состояние (определения, типы,…)
- Набор примитивных тактик, которые насыщают термы
- Тактики выбираются в зависимости от конструкций высокоуровневого языка
- В процессе выполняется *унификация* (решение уравнений) и *проверка типов* (контроль)
- Итогом является полностью сформированный терм (программа)

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 (2013). “Idris, a general-purpose dependently typed programming language: Design and implementation”. В: *Journal of Functional Programming* 23 (05), с. 552—593. ISSN: 1469-7653. DOI: 10.1017/S095679681300018X.


– (2017). *Type-Driven Development with Idris*. Manning Publications.


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