Управление вычислительными эффектами. Язык программирования Frank

Г. А. Лукьянов
georgiylukjanov@gmail.com

Южный федеральный университет
Институт математики, механики и компьютерных наук им. И. И. Воровича
Кафедра информатики и вычислительного эксперимента

14 декабря 2016

Содержание

Содержание


CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

Неформальное определение чистой функции

Прозрачность по ссылкам

Выражение *e* называется **прозрачным по ссылкам** (referential transparent), если для любой программы *p* каждое вхождение *e* в *p* может быть заменено на результат вычисления *e* без изменения результата вычисления *p*.

Неформальное определение чистой функции

Прозрачность по ссылкам

Выражение e называется **прозрачным по ссылкам** (referential transparent), если для любой программы p каждое вхождение e в p может быть заменено на результат вычисления e без изменения результата вычисления p.

Чистая функция

Функция f называется **чистой** если выражение $f(x)$ прозрачно по ссылкам для каждого прозрачного по ссылкам выражения x.

Примеры чистых и грязных функций

Чистая функция (Haskell)

```
add :: Int -> Int -> Int
add x y = x + y
```

Примеры чистых и грязных функций

Чистая функция (Haskell)

```
add :: Int -> Int -> Int
add x y = x + y
```

Чистая функция (C#)

```
static int Add (int x, int y) {
  return x + y;
}
```

Примеры чистых и грязных функций

Чистая функция (Haskell)

```
add :: Int -> Int -> Int
add x y = x + y
```

Чистая функция (C#)

```
static int Add (int x, int y) {
  return x + y;
}
```

Грязная функция (Haskell)

```
messyAdd :: IO Int
messyAdd = do
  x <- getInt
  y <- getInt
  return (x + y)
```

Примеры чистых и грязных функций

Чистая функция (Haskell)

```
add :: Int -> Int -> Int
add x y = x + y
```

Чистая функция (C#)

```
static int Add (int x, int y) {
  return x + y;
}
```

Грязная функция (Haskell)

```
messyAdd :: IO Int
messyAdd = do
  x <- getInt
  y <- getInt
  return (x + y)
```

“Чистая” функция (C#)

```
static int MessyAdd () {
  var x = ReadInt();
  var y = ReadInt();
  KillAllHumans();
  return x + y;
}
```

Подходы к типизации побочных эффектов (не является классификацией)

- Функторы
- Аппликативные функторы (идиомы)
- Монады
- Стрелки (arrows)
- Алгебраические эффекты и обработчики (handlers) эффектов

Монады

Интуиция

- Контейнер для значений с особыми правилами работы
- Последовательные вычисления с побочными эффектами и зависимостью между шагами

Монады

Интуиция

- Контейнер для значений с особыми правилами работы
- Последовательные вычисления с побочными эффектами и зависимостью между шагами

Монады в языках программирования

- LINQ в .NET
- Optinal в Swift
- Promise в JS
- Абстрактные монады в Haskell

Монады

Интуиция

- Контейнер для значений с особыми правилами работы
- Последовательные вычисления с побочными эффектами и зависимостью между шагами

Монады в языках программирования

- LINQ в .NET
- Optinal в Swift
- Promise в JS
- Абстрактные монады в Haskell

Проблемы

- Комбинирование эффектов

Алгебраических эффекты: интуиция

Аналогия с исключениями

- Сигнатуры (интерфейсы) эффектов — типы исключений
- Обработчики эффектов — обработчики исключений

Алгебраических эффекты: интуиция

Аналогия с исключениями

- Сигнатуры (интерфейсы) эффектов — типы исключений
- Обработчики эффектов — обработчики исключений

Аналогия с языками программирования

- Сигнатуры (интерфейсы) эффектов — абстрактный синтаксис
- Обработчики эффектов — интерпретаторы

Алгебраических эффекты: цели

Цели

- Разделение интерфейса и реализации
- Модульность
- Простота комбинирования эффектов

Содержание

Обзор языка

- *Строгость* по-умолчанию
- Алгебраические типы данных
- Явное отделение *типов-значений* и *типов-вычислений*
- *Сигнатуры эффектов*
- *Операторы* — обработчики эффектов
- Функции — тривиальные операторы, не обрабатывающие эффектов
- Полиморфизм эффектов на основе “окружающих эффектов” (ambient effects)

Значения и вычисления

Вычисление, возвращающее Char и не требующее эффектов

```
letterA [] Char
letterA = 'A'
```

Вызов `letterA`

```
main [] Char
main = letterA!
```

Значения и вычисления

Вычисление, возвращающее Char и не требующее эффектов

```
letterA [] Char
letterA = 'A'
```

Вызов letterA

```
main [] Char
main = letterA!
```

Вычисление, возвращающее вычисление, возвращающее Char

```
letterA' [] {[] Char}
letterA' = {'A'}
```

Вызов letterA'

```
main [] Char
main = letterA'!!
```

Значения и вычисления

Вычисление, возвращающее Char и не требующее эффектов

```
letterA [] Char
letterA = 'A'
```

Вызов `letterA`

```
main [] Char
main = letterA!
```

Вычисление, возвращающее вычисление, возвращающее Char

```
letterA' [] {[] Char}
letterA' = {'A'}
```

Вызов `letterA'`

```
main [] Char
main = letterA'!!
```

Вычисление с параметром, возвращающее Char

```
letterA'' Char [] Char
letterA'' c = c
```

Вызов `letterA''`

```
main [] Char
main = letterA'' 'A'
```

Алгебраические типы данных

Натуральные числа

```
data Nat
  = zero
  | suc Nat
```

Алгебраические типы данных

Натуральные числа

```
data Nat
  = zero
  | suc Nat
```

Равенство натуральных чисел

```
Nat == Nat [] Bool
zero  == zero  = tt
suc x == suc y = x == y
_     == z     = ff
```

Алгебраические типы данных

Натуральные числа

```
data Nat
  = zero
  | suc Nat
```

Равенство натуральных чисел

```
Nat == Nat [] Bool
zero  == zero  = tt
suc x == suc y = x == y
_     == z     = ff
```

Односвязный список

```
data List X
  = nil
  | X :: (List X)
```

Алгебраические типы данных

Натуральные числа

```
data Nat
  = zero
  | suc Nat
```

Равенство натуральных чисел

```
Nat == Nat [] Bool
zero  == zero  = tt
suc x == suc y = x == y
_     == z     = ff
```

Односвязный список

```
data List X
  = nil
  | X :: (List X)
```

Конкатенация списков

```
(List X) ++ (List X) [] List X
nil      ++ ys = ys
(x :: xs)++ ys = x :: (xs ++ ys)
```

Алгебраические типы данных

Натуральные числа

```
data Nat
  = zero
  | suc Nat
```

Равенство натуральных чисел

```
Nat == Nat [] Bool
zero  == zero  = tt
suc x == suc y = x == y
_     == z     = ff
```

Односвязный список

```
data List X
  = nil
  | X :: (List X)
```

Конкатенация списков

```
(List X) ++ (List X) [] List X
nil      ++ ys = ys
(x :: xs)++ ys = x :: (xs ++ ys)
```

Пара

```
data Pair A B = A & B
```

Тройка

```
data Triple A B C = A & B & C
```

Управляющие операторы

Условный оператор

```
if Bool then {[] X} else {[] X} [] X
if true  then t else e = t!
if false then t else e = e!
```

“Вычисляющий” условный оператор

```
cond Bool X X [] X
cond true  t f = t
cond false t f = f
```

Управляющие операторы

Условный оператор

```
if Bool then {[] X} else {[] X} [] X
if true  then t else e = t!
if false then t else e = e!
```

“Вычисляющий” условный оператор

```
cond Bool X X [] X
cond true  t f = t
cond false t f = f
```

“Монадическое” связывание

```
X >>= {X -> [] Y} [] Y
x >>= f = f x
```

Управляющие операторы

Условный оператор

```
if Bool then {[] X} else {[] X} [] X
if true  then t else e = t!
if false then t else e = e!
```

“Вычисляющий” условный оператор

```
cond Bool X X [] X
cond true  t f = t
cond false t f = f
```

“Монадическое” связывание

```
X >>= {X -> [] Y} [] Y
x >>= f = f x
```

Функция map (на самом деле mapM)

```
map {A -> [] B} (List A) [] List B
map f nil         = nil
map f (a :: as)   = f a :: map f as
```

Эффекты: сигнатура и интерпретация

Сигнатура эффекта исключений

```
sig Exception E
  = throw E [] {}
```

Эффекты: сигнатура и интерпретация

Сигнатура эффекта исключений

```
sig Exception E
  = throw E [] {}
```

Тип Maybe A

```
data Maybe X
  = just X
  | nothing
```

Тип Either A B

```
data Either A B = left A
                | right B
```

Эффекты: сигнатура и интерпретация

Сигнатура эффекта исключений

```
sig Exception E
  = throw E [] {}
```

Тип Maybe A

```
data Maybe X
  = just X
  | nothing
```

Тип Either A B

```
data Either A B = left A
                | right B
```

Интерпретация эффекта Exception с помощью типа Either E B

```
catchErr [Exception E ? X] [] Either E X
catchErr [x]                  =  right x
catchErr [throw e ? k]        =  left e
```

Эффекты: сигнатура и интерпретация

Сигнатура эффекта исключений

```
sig Exception E
  = throw E [] {}
```

Тип Maybe A

```
data Maybe X
  = just X
  | nothing
```

Тип Either A B

```
data Either A B = left A
                | right B
```

Интерпретация эффекта Exception с помощью типа Either E B

```
catchErr [Exception E ? X] [] Either E X
catchErr [x]                  =  right x
catchErr [throw e ? k]        =  left e
```

Интерпретация эффекта Exception с помощью типа Maybe A

```
catch [Exception E ? X] [] Maybe X
catch [x]                  =  just x
catch [throw e ? k]        =  nothing
```

Интерпретация Exception как Maybe

Безопасное вычитание натуральных чисел

```
Nat    - Nat [Exception ()] Nat
x      - zero   = x
zero   - y      = throw () {}
suc x  - suc y  = x - y
```

Интерпретация Exception как Maybe

Безопасное вычитание натуральных чисел

```
Nat    - Nat [Exception ()] Nat
x      - zero   = x
zero   - y      = throw () {}
suc x  - suc y  = x - y
```

Запуск вычисления

```
main [] Maybe Nat
main = catch ? 0 - 1
---------------------
$ ./frank examples.fk
nothing
```

Запуск вычисления

```
main [] Maybe Nat
main = catch ? 1 - 0
---------------------
$ ./frank examples.fk
(just 1)
```

Интерпретация Exception как Maybe

Список отменяемых вычислений, порождающих Nat

```
differences [] List {[Exception ()] Nat}
differences = {1 - 1} :: ({0 - 1} :: ({2 - 1} :: nil))
```

Интерпретация Exception как Maybe

Список отменяемых вычислений, порождающих Nat

```
differences [] List {[Exception ()] Nat}
differences = {1 - 1} :: ({0 - 1} :: ({2 - 1} :: nil))
```

Интерпретация списка вычислений

```
exec (List {[Exception ()] X}) [] List (Maybe X)
exec nil = nil
exec (action :: rest) = (catch ? action!) :: (exec rest)
```

Интерпретация Exception как Maybe

Список отменяемых вычислений, порождающих Nat

```
differences [] List {[Exception ()] Nat}
differences = {1 - 1} :: ({0 - 1} :: ({2 - 1} :: nil))
```

Интерпретация списка вычислений

```
exec (List {[Exception ()] X}) [] List (Maybe X)
exec nil = nil
exec (action :: rest) = (catch ? action!) :: (exec rest)
```

Запуск вычисления

```
main [] List (Maybe Nat)
main = exec (differences!)
---------------------
((just 0) :: (nothing :: ((just 1) :: nil)))
```

Изменяемое состояние

Сигнатура эффекта State

```
sig State S
  = get [] S
  | put S [] ()
```

Интерпретатор эффекта state

```
state S [State S ? X] [] X
state _ [x]             = x
state s [get   ? k]     = state s ? k s
state _ [put s ? k]     = state s ? k ()
```

Изменяемое состояние

Сигнатура эффекта State

```
sig State S
  = get [] S
  | put S [] ()
```

Интерпретатор эффекта state

```
state S [State S ? X] [] X
state _ [x]           = x
state s [get   ? k]   = state s ? k s
state _ [put s ? k]   = state s ? k ()
```

Пример: инкремент переменной

```
next [State Nat] Nat
next = put (suc (get!)) >>
       get!
```

Запуск вычисления

```
main [] Nat
main = state zero ? next!
-------------------------
(suc zero)
```

Вычисления в конфигурируемом окружении

Сигнатура эффекта Reader

```
sig Reader E
  = ask [] E
```

Интерпретатор эффекта state

```
reader E [Reader E ? X] [] X
reader _ [x]              = x
reader e [ask   ? k]  = reader e ? k e
```

Вычисления в конфигурируемом окружении

Сигнатура эффекта Reader

```
sig Reader E
  = ask [] E
```

Интерпретатор эффекта state

```
reader E [Reader E ? X] [] X
reader _ [x]              = x
reader e [ask   ? k]  = reader e ? k e
```

Пример: инкремент переменной

```
ex1 [Reader Nat] Nat
ex1 = ask! >>= {cfg -> suc cfg}
```

Запуск вычисления

```
main [] Nat
main = reader zero ? ex1!
-------------------------
(suc zero)
```

Комбинирование эффектов

Комбинирование Reader и State

```
reader_state_example [Reader Nat, State Nat] Nat
reader_state_example = ask! >>= {cfg -> next! >> get!}
```

Комбинирование эффектов

Комбинирование Reader и State

```
reader_state_example [Reader Nat, State Nat] Nat
reader_state_example = ask! >>= {cfg -> next! >> get!}
```

```
main [] Nat
main = reader zero ? state zero ? reader_state_example!
```

Вычисление числе Фибоначчи

Обёртка над вычислением с эффектом

```
nth_fib Nat [] Nat
nth_fib n = state (1 & 1 & n) ? fibs_state!
```

Вычисление числе Фибоначчи

Обёртка над вычислением с эффектом

```
nth_fib Nat [] Nat
nth_fib n = state (1 & 1 & n) ? fibs_state!
```

N-е число Фибоначчи, считая с нуля: вычисление с эффектом State

```
fibs_state [State (Triple Nat Nat Nat)] Nat
fibs_state = get! >>=
  { (x1 & x2 & zero) -> x1
  | (x1 & x2 & suc n) -> put (x2 & (x1 + x2) & n) >>
                         fibs_state!
  }
```

Вычисление числе Фибоначчи

Обёртка над вычислением с эффектом

```
nth_fib Nat [] Nat
nth_fib n = state (1 & 1 & n) ? fibs_state!
```

N-е число Фибоначчи, считая с нуля: вычисление с эффектом State

```
fibs_state [State (Triple Nat Nat Nat)] Nat
fibs_state = get! >>=
  { (x1 & x2 & zero) -> x1
  | (x1 & x2 & suc n) -> put (x2 & (x1 + x2) & n) >>
                         fibs_state!
  }
```

```
main [] Nat
main = nth_fib 4
-------------------------
5
```

Взаимодействие с внешним миром: эффект Console

Сигнатура эффекта (интерфейс) Console

```
sig Console
  = inch [] Char
  | ouch [] ()
```

Взаимодействие с внешним миром: эффект Console

Сигнатура эффекта (интерфейс) Console

```
sig Console
  = inch [] Char
  | ouch [] ()
```

Напоминание: функция map

```
map {S -> []T} (List S) [] List T
map f nil         = nil
map f (x :: xs) = f x :: map f xs
```

Взаимодействие с внешним миром: эффект Console

Сигнатура эффекта (интерфейс) Console

```
sig Console
  = inch [] Char
  | ouch [] ()
```

Напоминание: функция map

```
map {S -> []T} (List S) [] List T
map f nil         = nil
map f (x :: xs) = f x :: map f xs
```

Печать строки

```
main [Console] ()
main = map ouch ('h'::('e'::('l'::('l'::('o'::nil)))))
```

Комбинирование Console и State

Генерация чисел из $[1, n]$

```
range_1_n Nat [State (List Nat)] List Nat
range_1_n zero = get!
range_1_n (suc n) = get! >>= {
    xs -> put (suc n :: xs) >>
          range_1_n n
  }
```

Комбинирование Console и State

Генерация чисел из $[1, n]$

```
range_1_n Nat [State (List Nat)] List Nat
range_1_n zero = get!
range_1_n (suc n) = get! >>= {
    xs -> put (suc n :: xs) >>
          range_1_n n
  }
```

Печать диапазона

```
rangePrinter [Console, State (List Nat)] List ()
rangePrinter = (map ouch
  (concat (map natToString (range_1_n 3))))
```

Комбинирование Console и State

Генерация чисел из $[1, n]$

```
range_1_n Nat [State (List Nat)] List Nat
range_1_n zero = get!
range_1_n (suc n) = get! >>= {
    xs -> put (suc n :: xs) >>
          range_1_n n
  }
```

Печать диапазона

```
rangePrinter [Console, State (List Nat)] List ()
rangePrinter = (map ouch
  (concat (map natToString (range_1_n 3))))
```

Запуск вычисления

```
main [Console] ()
main = state nil ? rangePrinter!
```

Содержание

Выводы

- Концепции алгебраических эффектов и обработчиков эффектов находятся в стадии исследования и развития.
- Алгебраические эффекты и обработчики позволяют явно отделить интерфейс эффекта и его интерпретацию.
- Frank — прототип языка, включающего концепцию алгебраических эффектов в ядро: сигнатуры эффектов являются самостоятельной сущностью.

Материалы

- Do be do be do. Sam Lindley, Conor McBride, and Craig McLaughlin. To appear at POPL 2017.
- A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. J. Log. Algebr. Meth. Program., 84(1):108–123, 2015. URL `http://dx.doi.org/10.1016/j.jlamp.2014.02.001`.
- P. B. Levy. Call-By-Push-Value: A Functional/Imperative Synthesis, volume 2 of Semantics Structures in Computation. Springer, 2004.

- Слайды и исходные коды примеров `https://github.com/geo2a/frank-mmcs-seminar`