

ФП: введение, λ -исчисление, введение в Haskell

Пугачёв Константин (K.V.Pugachev@inp.nsk.su)

2018-11-06 18h

Вторая половина СТМияП

Функциональное программирование (около 4 лекций)

- Общие идеи
- Haskell от синтаксиса до некоторых монад
- Задачи: несколько простых на идею + одна реальная

Веб-программирование (около 3 лекций)

- Общие идеи
- Фичи ECMAScript2016..2018, не отмеченные в 1 части
- Работа с DOM, стилями, ...: API браузера
- ECMAScript на сервере; некоторые инструменты
- Задачи: несколько простых на идею + одна реальная

Формальности

- Не понял – задай вопрос.
- Сдано всё и вовремя, нет пропусков - 5.
- Сдано не всё/невовремя/2+ пропуска - <5.

Введение

Основные парадигмы программирования.

- Императивное программирование: “как решать задачу?”
 - процедурное, структурное, ОО, ...

```
var fac = n => {  
  var s = 1;  
  while(n) s *= n--;  
  return s;  
};
```

“чтобы посчитать факториал, **нужно...**”

- Декларативное программирование: “какой нужен результат?”
 - функциональное, логическое, ...

```
var fac = n => n > 0 ? n * fac(n-1) : 1;
```

“факториал – это...”

Идеи функционального программирования

- программа как комбинация функций
 - (vs императивное: комбинация процедур)
- чистые функции
 - нет побочных эффектов
 - результат зависит только от аргументов
- декларативность
 - описание результата, а не списка действий
- отсутствие изменяемого состояния

...и многое другое

Зачем изучать ФП

- возможность подойти к задаче с другой стороны
 - лучшее понимание шаблонов, STL, boost
 - лучшее понимание рекурсивных алгоритмов
- полезные идеи и инструменты
 - абстракции над циклами (map, fold, filter)
- более надёжный код
 - изменяемое состояние \equiv баги
 - если работают функция, то работает их суперпозиция
 - больше абстракций, меньше бойлерплейта
 - в теории многопоточность бесплатна
- мейнстримовые языки заимствуют идеи ФП
 - map, fold, filter (C++, python, Java, ECMAScript)
 - замыкания (python, Java, немножко C++, ECMAScript)
 - моноиды (Java)
 - монады (jQuery)

Лямбда-исчисление Чёрча

Лямбда-исчисление Чёрча: общий план

Вычисление – выражение, скомбинированное из функций.

- Все функции анонимны, именование – только для удобства (самоссылки невозможны)
- Все функции от одного аргумента
- Переменных, констант, имён нет или от них можно избавиться, есть только аргументы лямбд

С точки зрения вычислений

- λ -исчисление эквивалентно машине Тьюринга
- м. Тьюринга – ИП, возведённое в абсолют
- λ -исчисление – ФП, возведённое в абсолют

Лямбда-исчисление Чёрча: принципы

- Переменные (не меняются)

x, y, z, \dots

- Аппликация – применение функции к аргументу

Было	Стало
выражения f, x	$f x$

- Абстракция – создание функции из выражения

Было	Стало
выражение e , зависящее от x	$\lambda x.e$

Примеры лямбда-термов:

$x, \lambda x.x, f x, \lambda x.f x,$
 $\lambda y.\lambda x.f x, (\lambda f.\lambda x.f x) g y$

Лямбда-исчисление Чёрча: преобразования

- α -эквивалентность: переименование связанной (по которой абстрагировались) переменной

Пример: $\lambda x.x z \equiv \lambda y.y z$ (но $\lambda x.x z \neq \lambda z.z z$)

- β -редукция: подстановка аргумента

Было	Стало
$(\lambda x.f x) y$	$f y$
$(\lambda x.x x) (\lambda x.x)$	$(\lambda x.x)(\lambda x.x)$
$(\lambda x.x)(\lambda x.x)$	$\lambda x.x$
$(\lambda x.\lambda y.x y) y$	$\lambda z.y z \neq \lambda y.y y$

- η -преобразование: “снятие очевидной абстракции”

Было	Стало
$(\lambda x.x) x$	x
$(\lambda x.f x) x$	$f x$
$(\lambda x.f x) x, f = \lambda x x$	$\lambda x x$
$(\lambda x.f x) x, f = \lambda y x$	$?$

Нумералы Чёрча

Число – функция, которая к переданному значению применяет переданную функцию столько раз, чему оно равно.

$0 := \lambda f. \lambda x. x$ – применяет 0 раз f к x

$1 := \lambda f. \lambda x. f\ x$ – применяет 1 раз f к x

$2 := \lambda f. \lambda x. f\ (f\ x)$ – применяет 2 раза f к x

$\text{succ} := \lambda n. \lambda f. \lambda x. f\ (n\ f\ x)$ – навешиваем f на n ещё раз

$[2] \sim f\ (f\ x), \quad [2+1] \sim f\ [2] \sim f\ (f\ (f\ x))$

Нумералы Чёрча

$plus := \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)$ – навешиваем m раз f на n

$$[2] \sim f \ (f \ x), \quad [2+3] \sim f \ (f \ [3]) \sim f \ (f \ (f \ (f \ x)))$$

n – функция, n раз навешивает f .

$mult := \lambda m. \lambda n. \lambda f. m \ (n \ f)$ – навешиваем m раз функцию n .

$$[2] \sim f \ (f \ x), \quad [2*3] \sim 3 \ f \ (3 \ f \ x) \sim 3 \ f \ (f \ (f \ (f \ x))) \sim f \ (f \ (f \ (f \ (f \ (f \ x)))))$$

Степень нумералов Чёрча

$pow := \lambda b. \lambda e. e\ b$ – математика

ФВП, каррирование, понимание

Функция высшего порядка: функция, принимающая или возвращающая функции.

Каррирование: преобразование функции от многих аргументов к функции от одного аргумента.

```
const f = (x, y) => x + y;  
const f = x => y => x + y;
```

Дуализм бинарной функции (“полиизм” n -арной функции):

- f – функция от двух аргументов, преобразующая их в сумму
- f – функция от одного числа, преобразующего его в функцию добавления

```
const map = f => xs => xs.map(f);
```

Дуализм map:

- map – берёт функцию и массив, преобразующая элементы массива посредством функции
- map – берёт функцию над элементами и возвращает её модификацию, которая работает над массивом

Степень нумералов Чёрча

$pow := \lambda b. \lambda e. e \ b$ – математика

Дуализм нумерала Чёрча:

- НЧ принимает функцию и значение, возвращает многократное применение функции к значению
- НЧ берёт f и возвращает функцию, которая навешивает f несколько раз

3 – функция, которая берёт f и возвращает $f \circ f \circ f$.

2 – функция, которая берёт f и возвращает $f \circ f$.

$2 \ 3 \equiv 3 \circ 3$ – двойное навешивание тройки

$3 \ f \equiv f \circ f \circ f$ – навесили тройку первый раз

$3 \ (f \circ f \circ f) \equiv (f \circ f \circ f) \circ (f \circ f \circ f) \circ f \circ f \circ f \equiv 9 \ f$ – навесили тройку второй раз

Итого, $2 \ 3 \equiv 9$

Наркоз уже не тот

$pred := \lambda n. \lambda f. \lambda x. n (\lambda g. \lambda h. h (g f)) (\lambda u. x) (\lambda u. u)$

$sub := \lambda m. \lambda n. n \text{ pred } m$

«Если вы ничего не поняли, не огорчайтесь. Вычитание придумал Клини, когда ему вырывали зуб мудрости. А сейчас наркоз уже не тот»

Рекурсия невозможна?

Можно определить if , eq , mul , 1 , dec как некоторые λ -термы.

Тогда квадрат выражается так:

$$square := \lambda x.mul\ x\ x \equiv \lambda x.(\lambda lhs.\lambda rhs....)\ x\ x$$

и можно будет выразить всё как $\lambda x_1.\lambda x_1.\lambda x_3....$

Но что будет, если реализовать факториал?

$$fac := \lambda x.if\ (eq\ x\ 1)\ 1\ (mul\ x\ (fac\ (dec\ x)))$$

Нечестно. Факториал ссылается сам на себя, подстановка не может быть завершена.

Косвенная рекурсия? Нет, сводится к самовывозову после подстановки.

Комбинатор неподвижной точки

Неподвижная точка x функции f – точка, где $f(x) = x$.

```
> var x = 1;  
undefined  
> while (x !== Math.cos(x)) x = Math.cos(x);  
0.7390851332151607
```

Комбинатор неподвижной точки:

$fix(f) = x$, где $x = f(x)$ – запахло бесконечной рекурсией

Рекурсия без самовывозов

Y -комбинатор:

$$\underline{Y = \lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))}$$

Y вычисляет неподвижную точку функции, порождает бесконечную рекурсию без **самовывозов** каких-либо функций.

С помощью Y -комбинатора в λ -исчислении возможна рекурсия.

$fac\ n \equiv Y\ F$, где $F = \lambda f.\lambda n.if\ (iszero\ n)\ 1\ (mult\ n\ (f\ (pred\ n)))$

Нетипизированное и типизированное λ -исчисление

За типами не следят:

```
var xs = [1, [2]]; // OK
```

За типами следят:

```
std::vector<int> xs = {1, {2}}; // ERROR  
std::vector<std::vector<int>> xs = {1, {2}}; // ERROR
```

Типизация может быть и в λ -исчислении, накладывая свои условия.

Например, Y -комбинатор не работает в типизированном λ -исчислении.

Нетипизированное и типизированное λ -исчисление

В популярных ЯП можно использовать

- В JS, python – нетипизированное λ -исчисление
- В C++, Java – типизированное λ -исчисление
- В C++ с `void*`, в Java с `Object` – нетипизированное λ -исчисление

Haskell

Haskell (1990)

- назван в честь математика Хаскелла Карри (1900 – 1982)
 - Каррирование – тоже в честь него
 - Язык Curry – тоже в честь него
 - чистый функциональной ЯП
 - общего назначения
 - типизация строгая, статическая
 - вывод типов (Хиндли – Милнер), типы можно не указывать явно
 - все функции каррированы
 - ленивые вычисления
 - сопоставление с образцом
 - алгебраические типы данных
- и ещё много чего

Возможные инструменты, команды

- Notepad++
- Haskell Platform
 - the Glasgow Haskell Compiler
 - the Cabal build system
 - the Stack tool for developing projects
 - support for profiling and code coverage analysis
 - 35 core & widely-used packages

Пара команд:

```
ghc мой_файл          # компиляция программы
ghci                  # интерпретатор
cabal install пакет  # установка пакета
```


Информация и поиск

Самые простые объяснения:

- Денис Москвин. Функциональное программирование на языке Haskell – онлайн курс
- Денис Москвин. Функциональное программирование на языке Haskell (часть 2) – онлайн курс
- Miran Lipovača. Learn You a Haskell for Great Good! (eng) / Изучай Haskell во имя добра (рус)

Для тех, кто умеет читать:

- Документация
- Антон Холомьёв. Учебник по Haskell

Очень полезный поиск:

- Hoogle

Основные типы

```
a          -- какого-то тип

Int         -- короткое целое
Integer    -- длинное целое (длинная арифметика)
Float, Double -- с плавающей точкой
Char        -- символ
String      -- строка
Bool        -- логическое значение (True, False)

[a]         -- СПИСОК элементов какого-то типа
[Char]      -- СПИСОК символов (= String)
[[Int]]     -- СПИСОК СПИСКОВ коротких целых
```

Значения, функции

```
x :: Int -- указание типа  
x = 1
```

```
c = 'a' -- c будет Char, вывод типов работает
```

```
f x = x == c -- x будут требовать Char,  
              -- т.к. сравнивают с Char
```

```
add x y = x + y
```

```
five = add 2 3 -- применение функции
```

```
add3 = add 3 -- функция каррирована
```

```
name = "Haskell" -- String
```

```
Name = "Haskell" -- ошибка, значения – с маленькой буквы
```

Приоритеты операций

- 10 приоритетов
- ассоциативность (левая, правая, нет)
- применение функции имеет наивысший приоритет

Посмотрим информацию в интерпретаторе:

```
Prelude> :i (+)
class Num a where
    (+) :: a -> a -> a
    ...
    -- Defined in `GHC.Num'
infixl 6 +
Prelude> :i (&&)
(&&) :: Bool -> Bool -> Bool    -- Defined in `GHC.Classes'
infixr 3 &&
```

Итог: + – левоассоциативный, && – правоассоциативный, + приоритетнее &&.

Литералы чисел полиморфны

```
1      -- число какого-то числового типа
1.0    -- число какого-то нецелого типа

2 / 3      -- 0.6666666666666666
2.0 / 3.0  -- 0.6666666666666666

1 :: Int      -- число типа Int
1 :: Double   -- число типа Double
1.0 :: Int    -- ошибка
1.0 :: Double -- OK
```

Типы функций, кортежей

Кортеж (не путать с кортежем в python) – как структура, только поля не имеют имён, соответствует произведению типов.

Математика	Haskell
$f : R \times R \rightarrow R$	<code>f :: (Double, Double) -> Double</code>
$f(x, y) = x + y$	<code>f (x, y) = x + y</code> – функция от кортежа
$f : R \rightarrow R \rightarrow R$	<code>f :: Double -> Double -> Double</code>
$f(x) = g, \quad g(y) = x + y$	<code>f x = g where g y = x + y</code>
$f(x) = g, \quad g(y) = x + y$	<code>f x y = x + y</code>
$x : R \times Z \times \{True, False\}$	<code>x :: (Double, Int, Bool)</code> – кортеж

Функции

Математика	Haskell
$f(0) = 1$ $f(n) = n \times f(n-1)$	$f\ 0 = 1$ $f\ n = n * f\ (n-1)$ – несколько уравнений
$\lambda x. \lambda y. f\ x\ y$ $(f \circ g)(x) = f(g(x))$	$\backslash x \rightarrow \backslash y \rightarrow f\ x\ y$ $f\ .\ g = \backslash x \rightarrow f\ (g\ x)$

where, let in, охранные выражения

-- where, декларативный стиль

```
e = m * c^2 where  
  m = 12.0  
  c = 3e8
```

-- let in, композиционный стиль

```
e = let m = 12.0 -- let .. in  
    c = 3e8  
    in m * c^2
```

```
fac n | n <= 0    = 1 -- guards  
      | otherwise = n * fac (n-1)
```

Примечание: let .. in – выражение, where и охранные выражения – не выражения.