ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ*, НЕНОРМАЛЬНОЕ ПРОГРАММИРОВАНИЕ*, JAVASCRIPT*

Лямбда-исчисление на JavaScript

ibessonov 26 апреля 2017 в 19:06

Привет! В этой статье я хочу в очередной раз взглянуть на лямбда-исчисление. Теоретическую сторону вопроса на хабре обсуждали уже множество раз, поэтому взглянем на то, как лямбда-исчисление может выглядеть на практике, например, на языке JavaScript (чтобы примеры можно было выполнять прямо в браузере).

Итак, основная идея: всё есть функция. Поэтому мы ограничим себя очень узким кругом возможностей языка: любое выражение будет либо анонимной функцией с одним аргументом (`x => expr`), либо вызовом функции (`f (x)`). То есть весь код будет выглядеть похожим образом:

```
id = x => x
double = f => x => f (f (x))
```

Поскольку результатом работы функций будут другие функции, нам понадобится способ интерпретировать результат. Это единственное место, в котором пригодятся нетривиальные возможности JavaScript.

Базовые вещи

Начнём, как это принято, с условного ветвления. Введём две константы:

```
True  = t => f => t
False = t => f => f
```

Это функции «двух аргументов», `True` возвращает первый аргумент, `False` возвращает второй аргумент. То есть, справедливы следующие утверждения:

```
console.assert(1 == True  (1) (2))
console.assert(2 == False (1) (2))
```

Данные константы представляют логические истину и ложь, и позволяют написать функцию `If`:

```
If = b => t => f => b (t) (f)
console.assert(1 == If (True)  (1) (2))
console.assert(2 == If (False) (1) (2))
```

Это уже похоже на традиционный оператор `if`, просто синтаксически выглядит немного иначе. Имея условное ветвление, можно легко реализовать стандартные булевы операторы:

```
Not = x => If (x) (False) (True)
And = x => y => If (x) (y) (False)
Or  = x => y => If (x) (True) (y)
```

Следующим делом введём первую «структуру данных» — пару. Определение выглядит следующим образом:

```
Pair = x => y => (f => f (x) (y))
Fst  = p => p (True)
Snd  = p => p (False)

p = Pair (1) (2)
console.assert(1 == Fst (p))
console.assert(2 == Snd (p))
```

Оно выглядит немного страннее, но в нём есть смысл. Пара в данном случае есть функция, инкапсулирующая внутри себя 2 значения и способная передать их своему параметру при вызове. Подобным образом можно описать кортеж любой длины:

```
Triplet = x => y => z => (f => f (x) (y) (z))
Pentuplet = x => y => z => u => v => (f => f (x) (y) (z) (u) (v))
```

Вообще говоря, с подобным арсеналом мы уже могли бы определить `Byte` как кортеж из 8 логических значений, `Int` как кортеж из 4 `Byte` и реализовать на них машинную арифметику, но дело это рутинное и удовольствия никакого не принесёт. Есть более красивый способ описать натуральные числа — арифметика Чёрча.

Арифметика

Арифметика Чёрча описывает множество натуральных чисел с

нулём как функции от двух аргументов:

```
Zero  = s => z => z
One   = s => z => s (z)
Two   = s => z => s (s (z))
Three = s => z => s (s (s (z)))
```

Первый аргумент — это функция, второй аргумент — это что-то, к чему функция применяется `n` раз. Для их построения, по сути, нужны только ноль и функция `+1`:

```
Succ = n => (s => z => s (n (s) (z)))
```

`Succ` накидывает слева ещё один вызов `s` на уже имеющуюся цепочку вызовов, тем самым возвращая нам следующее по порядку натуральное число. Если данная функция кажется сложной, есть альтернативный вариант. Результат его работы будет абсолютно таким же, но накидывание `s` здесь происходит справа:

```
Succ = n => (s => z => n (s) (s (z)))
```

Тут стоит ещё описать способ преобразования чисел Чёрча во всем нам знакомый `int` — это в точности применение функции `x => x + 1` к нулю `n` раз:

```
toInt = n => n (x => x + 1) (0)

console.assert(0 == toInt (Zero))
```

```
console.assert(1 == toInt (One))
console.assert(2 == toInt (Two))
```

Аналогично определяются операции сложения, умножения и т.д.:

```
Add = n => m => m (Succ) (n)
Mul = n => m => m (Add (n)) (Zero)
Pow = n => p => p (Mul (n)) (One)
//↑↑ = n => m => m (Pow (n)) (One)
```

Таки образом можно продолжать реализовывать стрелочную нотацию, но смысла в этом нет: к этому моменту принцип работы с числами должен быть понятен.

Следующий шаг — вычитание. Следуя только что появившейся традиции, его реализация будет вот такой:

```
Sub = n => m => m (Pred) (n)
```

но остаётся проблемой реализация функции Pred. К счастью, Клини придумал её за нас.

```
Pred = n => Fst
    (n (p => Pair (Snd (p))
                  (Succ (Snd (p))))
       (Pair (Zero) (Zero)))
```

История гласит, что эта идея возникла у него во время приёма у дантиста, а анестезия тогда была посильнее, чем сейчас. Не буду с этим спорить, а объясню, что тут творится. Для получения

предыдущего числа мы строим пару `(n-1, n)` следующим образом: применяем `n` раз функцию `(x, y) -> (y, y+1)` к паре `(0, 0)` и у результата извлекаем левый компонент. Как следствие, легко заметить, что число, предыдущее для нуля, тоже будет нулём. Это спасает от неопределённости и даёт множество других преимуществ.

Для полноты картины приведу реализации операций сравнения:

```
IsZero = n => n (_ => False) (True)
Lte    = n => m => IsZero (Sub (n) (m))
Lt     = n => m => Lte (Succ (n)) (m)
Eq     = n => m => And (Lte (n) (m)) (Lte (m) (n))

Max = n => m => If (Lte (n) (m)) (m) (n)
Min = n => m => If (Lte (n) (m)) (n) (m)
```

Списки

Списки кодируются практически так же, как и натуральные числа — это тоже функции двух аргументов.

```
Nil = f => x => x
L1  = f => x => f (a0) (x)
L2  = f => x => f (a0) (f (a1) (x))
L3  = f => x => f (a0) (f (a1) (f (a2) (x)))
//...
```

Интересно отметить, что `False`, `Zero` и `Nil` являются одинаковыми функциями.

Если вы знакомы с функциональными операциями на списках, то,

вероятно, уже заметили, что эта структура описывает правую свёртку. Поэтому и реализуется она тривиально:

```
Foldr  = f => z => l => l (f) (z)
```

Теперь реализуем стандартные операции для персистентных списков — добавление в начало, получение «головы» и «хвоста» списка.

```
Append = a => l => (f => x => f (a) (l (f) (x)))
Head   = l => l (a => _ => a) ()

list = Append (1) (Append (2) (Nil))
console.assert(1 == Head (list))
```

Пустые скобки в конце описания `Head` — это голова пустого списка. В корректной программе данное значение никогда не должно быть использовано, поэтому я и выбрал синтаксически наименьшую величину. Вообще говоря, внутри этих скобок можно писать всё, что угодно. Позднее в статье будет ещё одно место, где я буду использовать пустые скобки по абсолютно той же причине.

Функция получения хвоста списка практически полностью повторяет функцию получения предыдущего натурального числа. По той же причине хвост пустого списка будет пустым списком.

```
Tail = l => Fst (
    l (a => p => Pair (Snd (p))
                      (Append (a) (Snd (p))))
      (Pair (Nil) (Nil))
```

```
)

console.assert(2 == Head (Tail (list)))
```

Как пример использования, приведу реализацию функции `map` и ещё некоторых других:

```
Map     = m => l => (f => x => l (a => f (m (a))) (x))
Length  = l => Foldr (_ => Succ) (Zero) (l)
IsEmpty = l => IsZero (Length (l))

// конвертация в стандартный список языка JavaScript
toList    = l => Foldr (x => y => [x].concat(y)) ([]) (l)
toIntList = l => toList (Map (toInt) (l))
function arraysEqual(a,b) { return !(a < b) && !(b < a); } // надо для ассертов
```

`Map` заменяет каждый элемент списка результатом вызова функции `f` на нём.
`Length` и `IsEmpty` говорят сами за себя. `toList` и `toIntList` будут полезны для тестирования и для вывода списков в консоль.

Рекурсия

К этому моменту работа исключительно с функциями стала подозрительно похожей на «нормальное» программирование. Пришло время всё испортить. Поскольку мы ограничили себя лишь объявлением и вызовом функций, у нас полностью отсутствует любой синтаксический сахар, а значит, многие простые вещи нам придётся записывать сложным образом.

Например, я хочу написать функцию `OnNonEmpty : fun =>`

`list => result`, которая будет вызывать функцию `fun` на списке `list` только если он не пуст. Попробуем:

```
OnNonEmpty = f => l => If (IsEmpty (l)) (Nil) (f (l))
```

Видите ошибку? Если `f` не останавливается на пустом списке, то и `OnNonEmpty` не остановится, хотя должна. Дело в том, что JavaScript предоставляет нам аппликативный порядок вычислений, то есть все аргументы функции вычисляются до её вызова, никакой ленивости. А оператор `If` должен вычислять лишь одну ветку в зависимости от условия. Поэтому функцию нужно переписать, и красивее она от этого не станет.

```
OnNonEmpty = f => l => (If (IsEmpty (l)) (_ => Nil) (_ => f (l))) ()
```

Теперь `If`, в зависимости от условия, возвращает функцию, которую нужно вычислить. И только после этого производится вычисление, только так мы можем гарантировать ленивость.

Вторая проблема — рекурсия. Внутри функции мы можем обращаться только к её формальным аргументам. Это значит, что функция не может ссылаться сама на себя по имени.

Но решение есть, это пресловутый «Комбинатор Неподвижной Точки». Обычно после этих слов приводится описание комбинатора Y, но для аппликативного порядка он не годится. Вместо него мы будем использовать комбинатор `Z`, бессовестно

подсмотренный в этой замечательной статье.

```
Z = f => (x => f (y => x (x) (y)))
         (x => f (y => x (x) (y)))
```

Комбинатор — это функция, выбираемая исходя из ровно одного свойства: `Z (f) == f (Z (f))`, то есть `Z(f)` — это такое значение `x`, что `x == f (x)`. Отсюда и термин «неподвижная точка». Но не нужно думать, что каким-то чудом комбинатор способен решать уравнения, вместо этого он представляет собой бесконечный рекурсивный вызов `Z(f) = f (f (f (f ( ... ))))`.

«Главное свойство» комбинатора даёт прекрасный «побочный эффект»: возможность реализации рекурсии. Например, запись:

```
MyFun = Z (myFun => ...)
```

эквивалентна записи:

```
MyFun = (myFun => ...) MyFun // если бы мы разрешили так делать
```

а значит первый формальный параметр анонимной функции фактически совпадает с самой функцией `MyFun`, и мы можем осуществлять в ней настоящие рекурсивные вызовы. Попробуем на примере поиска остатка от деления:

```
Rem = Z (rem => n => m => (
    If (Lt (n) (m)) (_ => n)
                    (_ => rem (Sub (n) (m)) (m))
) ())
console.assert(1 == toInt (Rem (Three) (Two)))
console.assert(0 == toInt (Rem (Three) (One)))
```

После этого можно реализовать наш первый полезный алгоритм — алгоритм Евклида. Забавно, но он вышел ничуть не сложнее поиска остатка от деления.

```
Gcd = Z (gcd => n => m => (
    If (IsZero (m)) (_ => n)
                    (_ => gcd (m) (Rem (n) (m)))
) ())
```

Последовательности

Последняя структура данных, которой я коснусь, это «бесконечные списки», или последовательности. Функциональное программирование славится возможностью работать с подобными объектами, и я просто не могу обойти их стороной.

Объявляются последовательности следующим образом:

```
Seq = head => tail => Pair (head) (tail)

SeqHead = seq => Fst (seq)
SeqTail = seq => (Snd (seq)) ()
```

Хвост последовательности в конструкторе — это функция вычисления новой последовательности, а не готовый результат.

Такой подход обеспечивает возможность генерировать хвост бесконечной длины.

Для возможности тестирования напишем функцию получения первых `n` элементов:

```
SeqTake = Z (take => n => seq => (
    If (IsZero (n)) (_ => Nil)
                    (_ => Append (SeqHead (seq))
                                 (take (Pred (n)) (SeqTail (seq))))
) ())
```

Если хотите поупражняться — реализуйте `SeqTake` без рекурсии. Это возможно, я гарантирую.

Теперь стоит привести пример какой-нибудь последовательности. Пусть это будут натуральные числа — всё-таки приходится работать только с тем, что уже реализовано:

```
Nat = (Z (natFrom => n => Seq (n) (_ => natFrom (Succ (n))))) (Zero)
console.assert(arraysEqual([0, 1, 2], toIntList (SeqTake (Three)
(Nat))))
```

Тут используется вспомогательная функция `natFrom (n)`, возвращающая последовательность натуральных чисел, начинающуюся с `n`. `Nat` — это просто результат `natFrom (Zero)`.

Осталось совсем немного, последние 2 функции, и они самые громоздкие из тех, что есть в данном тексте. Первая — функция

фильтрации последовательности. Она находит в передаваемой последовательности все элементы, удовлетворяющие переданному предикату:

```
SeqFilter = Z (filter => cond => seq => (
    If (cond (SeqHead (seq))) (_ => Seq (SeqHead (seq))
                                        (_ => filter (cond) (SeqTail
(seq))))
                              (_ => filter (cond) (SeqTail (seq)))
) ())
```

В случае, если головной элемент не выполняет предикат, `SeqFilter` возвращает отфильтрованный хвост. В противном случае это будет последовательность из текущей головы и всё того же отфильтрованного хвоста.

Вторая — последовательность простых чисел. Очередной вариант Решета Эратосфена в моём исполнении:

```
Primes = (Z (sieve => nums =>
    Seq (SeqHead (nums))
        (_ => sieve (SeqFilter (p => Not (IsZero (Rem (p) (SeqHead
(nums)))))
                               (SeqTail (nums)))
))) (SeqTail (SeqTail (Nat)))
```

Эту функцию можно назвать кульминацией статьи. Принцип её работы проще будет понять на псевдокоде:

```
sieve (nums) {
    p = head (nums)
    rest = tail (nums)
    return append (p, sieve (filter (n -> n % p != 0, rest)))
```

```
}
primes = sieve [2, 3, 4, ...]
```

Вот тест:

```
Ten = Mul (Two) (Add (Two) (Three)) // s => z => s (s (s (s (s (s (s (s (s (s (z))))))))))
console.assert(arraysEqual([2, 3, 5, 7, 11, 13, 17, 19, 23, 29],
                           toIntList (SeqTake (Ten) (Primes))))
```

Не знаю, как вас, а меня всё ещё удивляет, что подобные вещи возможно написать с помощью лишь чистых функций!

Заключение

В итоге у меня получилось такое вот странное введение в LISP на примере языка JavaScript. Надеюсь я смог показать, что абстрактные математические идеи на самом деле очень близки к реальности программирования. И после подобного взгляда лямбда-исчисление перестанет выглядеть чем-то чересчур академичным.

Ну и, конечно, вот ссылка на Github со всем кодом из статьи.

Спасибо!

Поделиться:

Сохранить:

Комментарии (52)

Похожие публикации

Функциональное программирование на Javascript

54

ПЕРЕВОД

mungobungo • 9 октября 2012 в 01:24

Функциональное программирование для всех

151

ПЕРЕВОД

sheknitrtch • 19 апреля 2012 в 11:45

Комбинатор неподвижной точки

9

steck • 30 декабря 2009 в 14:11

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета)

69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов 20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE 24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке 31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s 49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить

85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

139

marks • вчера в 14:19

Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16