Лекции по функциональному программированию

В. Н. Брагилевский

18 декабря 2010 г.

Содержание

1 Знакомство с языком Haskell

1.1 Императивное и функциональное программирование

Определим состояние программы (обозначаемое буквой σ) как множество значений всех ее переменных всех типов (τ) в некоторый момент времени:

$$\sigma = \{x_i \in \tau_i\}$$

Тогда процесс выполнения можно рассматривать как последовательность состояний от начального (исходных данных) до конечного, содержащего вычисленные результаты работы:

$$\sigma = \sigma_0 \to \sigma_1 \to \ldots \to \sigma_n = \sigma'$$

Изменение состояния, т.е. переход $\sigma_i \to \sigma_{i+1}$, выполняется посредством оператора присваивания:

$$v := Expr$$

Кроме того, в программах могут использоваться условные операторы (**if**) и операторы циклов (**for**, **while**), позволяющие организовывать ветвление и многократное выполнение некоторых действий.

Программирование, основанное на этих конструкциях, и соответствующие языки программирования называются *императивными*. Для выражения идеи «подалгоритмов» в язык могут также добавляться подпрограммы (процедуры и функции) и используемые в них глобальные переменные. Такое программирование обычно называют *процедурным*.

Тот же процесс изменения состояния от начального к конечному можно представлять несколько иначе, а именно, как функцию, ставящую в соответствие начальному состоянию, т.е. набору исходных данных (входным параметрам), конечное состояние, как набор результатов:

$$\sigma' = f(\sigma)$$

Таким образом, можно говорить, что любая программа — процесс вычисления результатов по исходным данным — это функция. Программирование и языки программирования, основанные на этой идее, называются *функциональными*. В функциональном программировании отсутствуют оператор присваивания, условный оператор как механизм ветвления (правда, обычно поддерживается *условное выражение* — аналог операции ?: в языке C), операторы циклов (заменяемые, как можно догадаться, рекурсией). Целью курса является изучение одного из функциональных языков — языка Haskell, названного так по имени американского математика и логика Хаскелла Карри (1900—1982).

Особый смысл программы как функции позволяет обходиться без создания исполняемого файла. При необходимости вычислить требуемое значение, достаточно запустить специальную программу («вычислитель») и

указать ей имя функции и её исходные данные. Такой способ работы называется интерпретацией, а сама программа-вычислитель — интерпретатором. Именно так изначально использовались функциональные языки, хотя в последствии компиляторы также появились.

1.2 Арифметические и логические выражения

Широко распространены два инструмента для работы с языком Haskell — это интерпретатор Hugs и компилятор GHC (Glasgow Haskell Compiler), включающий также интерпретатор ghci. Знакомство с языком можно начать с арифметических выражений, вычисляемых непосредственно в интерпретаторе.

Здесь и далее всё, что вводится и выводится в командной строке интерпретатора, будем обозначать так:

```
> 2 + 4
6
```

В строке, которая начинается с символа „>“, записываются вычисляемые выражения. Она называется командной строкой интерпретатора. После неё выводится результат вычислений.

В арифметических выражениях можно использовать сложение, вычитание, умножение и деление, возведение в степень (натуральную и произвольную вещественную), остаток и частное от деления:

```
> 2 + 4
6
> 2 * 3
6
> 4 - 23
-19
> 5 ^ 3
125
> 9 ** 0.5
3.0
> -4 + 3
-1
> 2 * (4 + (-3) + 1)
4
> 10 `div` 3
3
> 10 `mod` 3
1
```

В выражениях также поддерживаются сравнения, логические константы ($True$ и $False$) и логические операции:

```
> 3 < 5
True
> 8 == 9
False
> 8 /= 9
True
> 8 <= 9
True
> True && False
False
> True || False
True
> not False
True
```

Особенный интерес представляет операция вызова функции и расстановка скобок при таком вызове:

```
> sin 1.5707963
1.0
> sin (2 * 1.5707963)
5.35897931700572e-08
> asin (sin 1)
1.0
> abs (-1)
1
```

Такая расстановка скобок означает, что вызов функции имеет наивысший приоритет, поэтому если параметр функции сам по себе должен вычисляться, то его необходимо заключить в скобки.

Арифметические операции тоже можно рассматривать как функции и писать их слева от аргументов, заключая в скобки. Например, смысл следующих операций одинаков:

```
> (+) 2 3
5
> 2 + 3
5
```

Позже будет видно, что функции нескольких переменных также не требуют при вызове наличия скобок у своих параметров.

Все значения, участвующие в выражениях, принадлежат к определённому типу. В языке Haskell определены несколько *примитивных* типов данных: **Integer** (целые числа произвольного размера), **Int** (ограниченные четырёхбайтовые целые числа), **Bool**, **Char**, **Double**. В интерпретаторе можно выяснить тип любого выражения с помощью команды `:type выражение` (или `:t выражение`):

```
> :type 8 /= 9
8 /= 9 :: Bool
> :t 2+3
2+3 :: (Num t) => t
> (sin 4)^2 + (cos 4)^2
(sin 4)^2 + (cos 4)^2 :: (Floating t) => t
```

В двух последних примерах интерпретатор не может точно указать тип выражения. Вместо этого утверждается, что тип выражения 2+3 может быть любым числовым типом (принадлежит классу типов **Num**), а тип последнего выражения принадлежит к классу типов **Floating**, т.е. может оказаться любым типом с вещественными значениями.

1.3 Определение функций

Ясно, что одного только интерпретатора выражений для программирования недостаточно. Необходимо где-то писать код функций. Для этой цели служат текстовые файлы с расширением .hs. Приведем пример одного такого файла, содержащего объявление константы:

```
{--
   Пример файла first.hs.
--}

-- константа answer
answer = 42
```

Чтобы воспользоваться этой константой, необходимо загрузить[1] файл в интерпретатор:

```
> :load first.hs
> answer
42
```

Теперь всё готово для объявления первой функции, которая возводит в квадрат переданный ей аргумент:

```
sqr a = a * a
```

Вызов этой функции выполняется следующим образом:

```
> sqr 5
25
```

Можно заметить, что объявленная ранее константа — это всего-навсего функция без аргументов. Приведём примеры еще нескольких простых функций:

[1] Будем считать, что интерпретатор запущен из каталога, в котором находится файл first.hs.

```
double x = x * 2
cube a = a * a * a
cube1 a = a * sqr a
cube2 a = a^3
tg x = sin x / cos x
trig_test x = (sin x)^2 + (cos x)^2
f1 x = x^3 + 6*x^2 - 7*x + 2
f2 x = (sin (x+1) + log (x-1)) / tg x
```

Функции могут иметь несколько аргументов, при этом их объявление выглядит точно так же:

```
rect_square a b = a * b
mean x y = (x + y) / 2
mean_geom x y = sqrt (x*y)
```

А вот как выглядит их вызов:

```
> rect_square 2 5
10
> mean 3 4
3.5
> mean_geom 12 3
6.0
```

1.3.1 Условные выражения

Иногда требуется выбирать одно из двух значений в зависимости от выполнения некоторого условия. Для этого в языке Haskell используются условные выражения. Определим с их помощью две функции — вычисление наименьшего из двух чисел[2] и вычисление знака числа:

```
min1 a b = if a < b then a else b

sign z = if z < 0
         then -1
         else if z == 0
              then 0
              else 1
```

В отличии от условного оператора в выражении **if**/**then**/**else** часть **else** обязательна.

С помощью условного выражения и рекурсии можно вычислить значение факториала или числа Фибоначчи:

[2]Эта функция называется min1 вместо традиционного **min**, так как есть стандартная функция с таким именем. Здесь и далее мы избегаем совпадения имён собственных функций с именами функций из стандартной библиотеки. Суффиксы 1, 2 и 3 будут также использоваться для функций с одинаковым смыслом, но выраженных различными синтаксическими конструкциями.

```
fact n = if n == 0 then 1 else n * fact (n-1)

fib n = if n == 1 || n == 2
           then 1
           else fib (n-1) + fib (n-2)
```

1.3.2 Охранные выражения

Выбор значения функции по некоторому условию является настолько часто используемой операцией, что для неё придуман специальный синтаксис, который называется *охранными выражениями* (*guards*):

```
min2 a b | a < b = a
         | otherwise = b

sign1 z | z < 0 = -1
        | z == 0 = 0
        | otherwise = 1

fact1 n | n == 0 = 1
        | otherwise = n * fact1 (n-1)

fib1 n | n == 1 = 1
       | n == 2 = 1
       | otherwise = fib1 (n-1) + fib1 (n-2)
```

При вычислении значения функции здесь выбирается первое подходящее условие и соответствующее ему выражение. Условия, вообще говоря, не обязаны быть взаимоисключающими. Ключевое слово **otherwise** пишется в последнем варианте и означает „во всех остальных случаях“. Этого условия может и не быть. Если в процессе вычислений ни один вариант условия не подойдёт, будет выдано сообщение об ошибке. Синтаксис охранных выражения заимствован из математики, там он используется для обозначения элементов множества, удовлетворяющих некоторому условию, например:

$$\mathbb{N} = \{n \,|\, n \in \mathbb{Z},\, n > 0\}.$$

1.3.3 Сопоставление с образцом

Ещё один способ определения функции называется *сопоставлением с образцом* (*pattern matching*). Он заключается в явном указании значения функции от конкретного значения аргумента. Возьмём, к примеру, математическое определение факториала:

$$\begin{aligned} 0! &= 1, \\ n! &= n \cdot (n-1)! \end{aligned}$$

Совершенно аналогично можно определить функцию fact:

```
fact2 0 = 1
fact2 n = n * fact2 (n - 1)
```

При вычислении значения функции отдельные строки её определения[3] анализируются сверху вниз, причём подбирается подходящее. Последний вариант определения обычно является самым общим, подходящим в большинстве случаев.

При таком подходе функция вычисления чисел Фибоначчи определяется следующим образом:

```
fib2 1 = 1
fib2 2 = 1
fib2 n = fib2 (n-1) + fib2 (n-2)
```

Приведём два примера очевидно бесполезных функций, определённых посредством механизма сопоставления с образцом:

```
f 0 = 0
f 1 = 42
f 2 = 87
f n = n*2

g 0 = 0
g 1 = 42
g 2 = 87
g _ = -1
```

В функции g в качестве параметра в последней строке определения используется символ „_“, который сигнализирует, что в теле функции имя параметра не требуется. Эта строка будет выбрана при вычислении значения функции, если фактический параметр не будет соответствовать ни одному из перечисленных выше вариантов.

1.3.4 Типы функций

Во всех предыдущих примерах тип параметров не указывался, он *выводился* (*infer*) автоматически. Результат вывода типа можно узнать в интерпретаторе. Рассмотрим, к примеру, тип функции fact:

```
fact 0 = 1
fact n = n * fact (n - 1)
```

```
>:type fact
fact :: (Num t) => t -> t
```

Запись fact :: (**Num** t) => t -> t означает, что функция принимает в качестве параметра значение типа t и возвращает значение того же типа, где тип t может быть любым числовым типом. Настолько широкое обозначение

[3] Иногда их называют *предложениями* или *клозами* (*clause*).

типа может приводить к недоразумениям, например, попытка вычислить факториал от 0.5 приведёт к бесконечной последовательности рекурсивных вызовов. Поэтому тип параметров функции можно указать явно:

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

```
> :t fact
fact :: Integer -> Integer
```

Запись **Integer** −> **Integer** означает, что функция принимает один целочисленный параметр и возвращает целочисленное же значение.

Тип функции с несколькими параметрами указывается следующим образом:

```
rect_square :: Double -> Double -> Double
rect_square a b = a * b
```

В строке **Double** −> **Double** −> **Double** первые два типа **Double** обозначают типы параметров, а последний — тип результата.

Понятно, что тип параметров не обязан совпадать с типом результата, например:

```
odd1 :: Integer -> Bool
odd1 n = n `mod` 2 /= 0

even1 :: Integer -> Bool
even1 n = not (odd1 n)
```

1.3.5 Локальные объявления

Иногда в вычислениях некоторые выражения встречаются несколько раз, в таких случаях необходимы *локальные объявления*, которые позволяют избежать повторного вычисления их значения. Например, при вычислении значения функции

$$f(x) = 2(x-3)^4 + 4(x-3)^2 - 1$$

может быть полезно предварительно вычислить $(x-3)^2$. В следующем определении функции именно это и делается:

```
f1 x = let a = (x-3)^2 in
           2*a^2 + 4*a - 1
```

Конструкция **let**/**in** вводит локальное объявление a = (x−3)^2, которое можно использовать в последующих вычислениях. Область действия локального объявления ограничивается самой конструкцией **let**/**in**.

Подобным образом используется конструкция **where**:

```
f2 x = 2*a^2 + 4*a - 1
       where a = (x-3)^2
```

Покажем один пример использования локальных объявлений для вычисления значения функции

$$g(x,y) = \frac{x+y}{x-y} + (x+y)^2 + (x-y)^2.$$

```
g1 x y = let a = x+y
             b = x-y
         in a/b + a^2 + b^2

g2 x y = a/b + a^2 + b^2
         where
             a = x+y
             b = x-y
```

1.4 Пары, кортежи и списки

До сих пор использовались данные примитивных типов. В языке Haskell есть механизмы для построения производных типов. Самые важные из них — *кортежи* и *списки*.

Кортеж — это структура данных, содержащая фиксированное количество данных произвольных типов. Структуры данных, состоящие из данных произвольных типов, называют *гетерогенными*. Кортеж с двумя элементами называется *парой*. Список — это структура данных, содержащая произвольное (в т.ч. и бесконечное) количество данных одного типа. Структуры данных, состоящие из данных одного типа, называют *гомогенными*.

Объявление пары выглядит следующим образом:

```
vector = (1.8, 2.3)
info = ('a', 15)
```

Для работы с парами можно использовать вспомогательные функции **fst** и **snd**, они возвращают первый и второй элементы пары. Предположим, что пара вещественных чисел соответствует координатам вектора в прямоугольной системе координат. Вычислим длину вектора и координаты суммы векторов:

```
vector_length v = sqrt ((fst v)^2 + (snd v)^2)

vector_sum v1 v2 = (fst v1 + fst v2, snd v1 + snd v2)
```

Вызывать эти функции можно так:

```
> vector_length (3,4)
5.0
> vector_sum (1, 2) (3, 4)
(4,6)
```

Для обработки пар можно также использовать сопоставление с образцом:

```
vector_length1 (x,y) = sqrt (x^2 + y^2)

vector_sum1 (x1,y1) (x2,y2) = (x1+x2, y1+y2)
```

Важно понимать, что эти функции *по-прежнему* принимают в качестве параметров пары, поэтому порядок вызова этих функций не меняется:

```
> vector_length1 (3, 4)
5.0
> vector_sum1 (1, 2) (3, 4)
(4,6)
```

Так же как и раньше, при сопоставлении с образцом можно указывать конкретные значения. Например, определим функцию, которая возвращает истинное значение, если оба элемента пары являются нулевыми:

```
is_zero (0, 0) = True
is_zero (_, _) = False
```

Функция, которая возвращает истину, если второй элемент пары равен нулю:

```
snd_zero (_, 0) = True
snd_zero (_, _) = False
```

Ту же функцию можно определить немного иначе:

```
snd_zero1 (_, n) = n == 0
```

Знак подчеркивания при сопоставлении с образцом, как и прежде, означает, что соответствующее значение в теле функции не используется.

Сопоставление с образцом — это единственный способ для обработки кортежей, содержащих более двух элементов:

```
tuple = ('a', 12, True, 123)

process_tuple::(Char,Integer,Bool,Integer)
                    ->(Integer,Char,Bool)
process_tuple (ch, n1, cond, n2)
                      = (n1+n2, ch, not cond)
```

Функция process_tuple складывает второй и четвёртый элементы заданного кортежа, инвертирует третий элемент — логическое значение — и возвращает результат в виде нового кортежа уже из трёх элементов. Здесь можно заметить, как в объявлении типа функции участвуют кортежи.

Объявление списка выглядит так:

```
numbers = [1, 56, 3, 42]
```

Это объявление по смыслу совпадает со следующим:

```
numbers1 = 1:56:3:42:[]
```

Здесь используется операция (:), которая присоединяет элемент к уже существующему списку. Присоединённый элемент становится *головой* списка, а вся остальная часть списка — его *хвостом*. В конце последнего определения стоит пустой список, обозначаемый []. В принципе, вполне можно было присоединить несколько элементов к непустому списку:

```
numbers2 = 1:56:[3,42]
```

Строки являются списками символов, поэтому следующие определения полностью эквивалентны:

```
hello = "Hello"
hello1 = ['H','e','l','l','o']
hello2 = 'H':'e':'l':'l':'o':[]
```

При работе со списками можно пользоваться вспомогательными функциями **head** и **tail** которые возвращают голову и хвост непустого списка. Понятно, что основной метод обработки списков — рекурсивные вызовы. Например, вычислим сумму элементов списка:

```
sum1 [] = 0
sum1 list = head list + sum1 (tail list)
```

Две строки определения соответствуют случаям пустого и непустого списка.

Функция **sum** вызывается следующим образом:

```
> sum1 [1,2,4,9]
16
> sum1 []
0
```

Как и при обработке пар, сопоставление с образцом позволяет избежать явного обращения к функциям определения головы и хвоста списка:

```
sum2 [] = 0
sum2 (x:xs) = x + sum2 xs
```

Образец (x:xs) означает, что x соответствует голове списка, а xs — хвосту.

Сопоставление списков с образцами не обязательно выделяет только голову и хвост, для примера напишем функцию, вычисляющую сумму первого и второго элементов списка:

```
sum_first_two (x1:x2:xs) = x1 + x2
```

Вот её вызов:

```
> sum_first_two [1,2,4,9]
3
```

Функция может формировать новый список на основе списка, переданного ей в качестве параметра:

```
add_zero list = 0 : list

double_list list = list ++ list
```

Операция (++) — это операция конкатенации списков (и, естественно, конкатенация строк).

Можно также возвращать пару, содержащую исходный список и количество элементов в нём (стандартная функция **length** возвращает количество элементов в заданном списке):

```
combine_list_length list = (list, length list)
```

Вызовем функцию combine_list_length:

```
> combine_list_length [1,2,3,4,5]
([1,2,3,4,5],5)
```

В качестве ещё одного примера функции обработки списков можно привести функцию вычисления суммы нечётных элементов списка (стандартная функция **odd** возвращает истину, если переданное ей число нечётное):

```
odd_sum [] = 0
odd_sum (x:xs) | odd x = x + odd_sum xs
               | otherwise = odd_sum xs
```

Вычисление суммы нечётных элементов среди оставшихся элементов списка можно было бы оформить в виде локального объявления, например:

```
odd_sum1 [] = 0
odd_sum1 (x:xs) | odd x = x + rest
                | otherwise = rest
               where rest = odd_sum1 xs
```

Тип данных в объявлениях функций, работающих со списками, указывается так:

```
sum1 :: [Integer] -> Integer
```

Запись [**Integer**] означает, что в качестве параметра функция принимает список целых чисел.

1.5 Пример: решение квадратного уравнения

Рассмотрим пример вычисления решения квадратного уравнения. Будем считать, что по заданным трём вещественным коэффициентам функция возвращает список корней. Этот список будет пустым, если вещественных корней нет, содержать один элемент, если имеется только один кратный корень, и два элемента в остальных случаях.

Приведём полное решение, а затем немного его прокомментируем:

```
solve_square_equation :: Double -> Double -> Double
                              -> [Double]
solve_square_equation 0 _ _ =
        error "Уравнение не является квадратным"
solve_square_equation a b c
  = find_roots discr
    where
     discr = b*b - 4*a*c
     find_roots d
      | d < 0      = []
      | d == 0     = [-b/2/a]
      | otherwise = let sqrtd = sqrt d
                        a2 = 2*a
                    in [(-b - sqrtd)/a2,
                        (-b + sqrtd)/a2]
```

В этом примере локально объявляются не только константы, но и функции. В частности, функция find_roots, объявленная внутри функции, вычисляет список корней уравнения по заданному дискриминанту. Следует обратить внимание на то, что эта вложенная функция свободно пользуется параметрами исходной функции, т.е. область действия параметров функции распространяется на все локальные объявления.

Стандартная функция **error** выводит сообщение об ошибке и останавливает выполнение функции. В данном случае она вызывается, если старший коэффициент уравнения равен нулю, т.е. уравнение не является квадратным.

Пользуясь функцией решения квадратного уравнения, можно написать функцию, находящую решения набора квадратных уравнений, заданных тройками коэффициентов. Следующая функция принимает в качестве параметра список троек (кортежей из трёх элементов) и возвращает список списков корней уравнений:

```
solve_square_eqs :: [ (Double, Double, Double) ]
                          -> [ [Double] ]
solve_square_eqs [] = []
solve_square_eqs ((a,b,c):eqs) =
    solve_square_equation a b c : solve_square_eqs eqs
```

Ниже показан пример использования разработанных функций:

```
> solve_square_equation 1 (-5) 6
[2.0,3.0]
> solve_square_eqs [(1,2,5),(1,2,1),(1,-8,15)]
[[],[-1.0],[3.0,5.0]]
```

1.6 Чистые функции и ввод-вывод

Все написанные ранее функции являлись *чистыми*. Чистота функции означает, что результат её вызова зависит исключительно от значений параметров. Чистая функция, вызываемая с одним и тем же набором параметров, всегда возвращает один и тот же результат. К сожалению, в программировании зачастую оказывается, что этого недостаточно. Самый важный пример нарушения свойства чистоты — это ввод-вывод. Если внутри функции выполняется ввод-вывод, то её результат зависит ещё и от того, что именно введёт пользователь, или что будет прочитано из файла. Говорят также, что при вызове такой функции возникают *побочные эффекты*.

В языке Haskell предусмотрены механизмы, позволяющие отделить части программы, связанные с вводом-выводом, от чистых функций. Среди прочего для этой цели используются **do**-блоки. Рассмотрим пример функции, запрашивающей имя пользователя и приветствующей его:

```
hello = do
  putStr "Enter your name: "
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

Приведём пример выполнения этой функции:

```
> hello
Enter your name: Jane
Hello, Jane!
```

Стандартные функции **putStr**, **putStrLn** и **getLine** не являются чистыми, поэтому синтаксис их вызова отличается от того, что показывалось ранее. В частности, функции вывода ничего не возвращают, мы ими пользуемся только ради побочных эффектов (в данном случае для вывода на консоль). Поэтому их можно вызывать последовательно, что и делается в данном примере. Это возможно только в блоке **do**. Значение, возвращаемое функцией **getLine** должно быть сохранено, для этой цели здесь применяется оператор $<-$. Важно понимать, что на самом деле результатом функции **getLine** является не строка, в этом нетрудно убедиться:

```
> :type getLine
getLine :: IO String
```

Оператор $<-$ как раз и позволяет извлечь строковое значение из результата, возвращаемого функцией.

В следующем примере демонстрируются отличия, возникающие при вызове чистых функций из **do**-блока. Здесь вводится число и выводится его квадрат.

```
formatSquare a = show a ++ "^2 = " ++ show (a^2)

square = do
```

```
putStr "Enter number: "
numberStr <- getLine
let number = read numberStr
putStrLn (formatSquare number)
```

Стандартная функция **show** преобразует данные любых типов к строковому виду, а функция **read** выполняет обратное преобразование. Это чистые функции, как и написанная нами formatSquare. Для сохранения результата вызова таких функций внутри **do**-блока используется конструкция **let**. Покажем результат вызова функции square:

```
> square
Enter number: 5
5^2 = 25
```

1.7 Функция main и компиляция программы

Разумеется, интерпретация — не единственный способ исполнения программы на языке Haskell. Программу можно скомпилировать, получив обычный исполняемый файл, а затем запустить его на исполнение. Для этого в программе должна присутствовать функция main, это предопределённое название для функции, с которой начинается выполнение программы на языке Haskell.

Следующий пример представляет собой классическую программу «Привет, мир» на языке Haskell:

```
main = putStrLn "Hello, world"
```

Предположим, что эта программа сохранена в файле hello.hs. Скомпилировать и запустить её на исполнение можно в консоли:

```
$ ghc hello.hs -o hello
$ ./hello
Hello, world
```

2 λ-исчисление как формальная система

Одной из важнейших черт функционального программирования является использование функций как значений первого класса («first class value»). Это означает, что функции наряду с данными других типов — числами, логическими значениями, списками — могут быть параметрами или возвращаемыми значениями других функций, а кроме того с ними можно выполнять различные операции, например, композицию. Такое широкое применение функций требует введения удобной нотации (способа обозначения).

Попробуем определить смысл выражения $x - y$. Во-первых, его можно трактовать как функцию переменной x:

$$f(x) = x - y,$$

что иначе можно записать как

$$f : x \mapsto x - y.$$

С другой стороны то же самое выражение можно понимать как функцию переменной y:

$$g(y) = x - y$$

или

$$g : y \mapsto x - y.$$

Поскольку в этих примерах смысл выражения с точки зрения функции меняется кардинальным образом, нам необходим способ, позволяющий явно указать, что именно является аргументом функции, но при этом желательно избежать ее именования[4].

Соответствующая нотация была изобретена в 30-е годы XX века американским логиком и математиком Алонсо Чёрчем (1903—1995). В этой нотации в качестве вспомогательного символа используется греческая буква λ. Предыдущие примеры в λ-нотации Чёрча выглядят так:

$$f = \lambda x\,.\,x - y,\ \ g = \lambda y\,.\,x - y.$$

Вообще говоря, в именах f и g теперь необходимости нет, в выражениях их можно опускать:

$$\begin{array}{rcl}(\lambda x\,.\,x - y)(42) & = & 42 - y,\\ (\lambda y\,.\,x - y)(42) & = & x - 42.\end{array}$$

Если необходимо использовать функции многих переменных, λ-нотацию нетрудно расширить, записав, например:

$$h(x, y) = x - y = \lambda xy\,.\,x - y.$$

[4] Действительно, ведь в арифметике для указания того факта, что $2 \times 3 = 6$ не требуется вводить имена для чисел 2, 3 и 6.

Однако этого можно избежать, заменив их на функции одного переменной, значениями которых являются другие функции. Определим функцию h^* следующим образом:

$$h^* = \lambda x\,.\,(\lambda y\,.\,x - y).$$

Значением функции h^* в точке a будет уже другая функция:

$$h^*(a) = (\lambda x\,.\,(\lambda y\,.\,x - y))(a) = \lambda y\,.\,a - y.$$

Вычислим теперь $(h^*(a))(b)$:

$$(h^*(a))(b) = (\lambda y\,.\,a - y)(b) = a - b = h(a, b).$$

Таким образом, можно считать, что функция h^* *представляет* ту же функцию, что и h, но является функцией одной переменной. Такой способ записи функций многих переменных называется *каррированием* по имени уже знакомого нам Хаскелла Карри[5], имя которого также дало название языку программирования Haskell. Каррирование полезно еще и по той причине, что оно явно вводит в рассмотрение *функции высшего порядка* (*higher-order functions*), т.е. функции, аргументами или значениями которых среди прочего являются функции.

Введённая Чёрчем λ-нотация важна не сама по себе, а как инструмент построения формальной системы λ-исчисления, которое является теоретической основой функционального программирования.

2.1 Определение λ-термов

Определение 1 *Пусть задано счётное множество переменных x, y, z, ... Множество выражений, называемых λ-термами, можно определить с помощью следующих трёх условий:*

- *Все переменные являются λ-термами.*
- *Если M и N — произвольные λ-термы, то (MN) — λ-терм, называемый применением (application).*
- *Если x — переменная, а M — произвольный λ-терм, то $(\lambda x\,.\,M)$ — λ-терм, называемый абстракцией (abstraction).*

Рассмотрим примеры λ-термов:

(1) x

(2) (xy)

(3) $(\lambda x\,.\,x)$

[5]Вообще говоря, до Карри этим способом пользовались еще Моисей Исаевич Шёнфинкель (1889—1942) и Готлоб Фреге (1848—1925), но согласно широко известной, хотя и не слишком корректной шутке, легко понять, почему соответствующие названия не прижились.

(4) $((\lambda x\,.\,y)\,t)$

(5) $(\lambda x\,.\,y)$

(6) $(\lambda x\,.\,(\lambda y\,.\,(xy)))$

(7) $((\lambda x\,.\,(\lambda x\,.\,(xu)))(\lambda x\,.\,v))$

Здесь терм (2) представляет собой применение переменной x к переменной y, терм (3) можно трактовать как тождественную функцию, а терм (5) — как константную (ее значение всегда равно y, независимо от значения аргумента x). Терм (7) — это применение одной абстракции к другой; здесь можно обратить внимание на многократное использование переменной x в λ-абстракциях, наше определение этому не препятствует, хотя при понимании таких выражений возможны определённые трудности.

Заметим, что сам символ λ термом не является, он используется только как признак построения λ-абстракции.

Прежде чем приступить к изучению свойств λ-термов, примем несколько соглашений, упрощающих соответствующие записи и рассуждения. Во-первых, в качестве переменных всюду будем использовать буквы x, y, z, t, u, v или w с индексами и без, а прописными латинскими буквами M, N, P и Q будем обозначать произвольные λ-термы.

Во-вторых, в выражениях с λ-термами будем опускать все внешние скобки, а также некоторые внутренние, считая, что:

- Применение является левоассоциативным, т.е. выражение MNP понимается как $((MN)P)$.
- Абстракция является правоассоциативной, причем символ λ захватывает максимально большое выражение, т.е. терм $\lambda x\,.\,MN$ соответствует $(\lambda x\,.\,(MN))$, а терм $\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,M$ означает $(\lambda x\,.\,(\lambda y\,.\,(\lambda z\,.\,M)))$.

В соответствии с принятыми соглашениями приведённые выше примеры можно переписать следующим образом:

(1) x

(2) xy

(3) $\lambda x\,.\,x$

(4) $(\lambda x\,.\,y)\,t$

(5) $\lambda x\,.\,y$

(6) $\lambda x\,.\,\lambda y\,.\,xy$

(7) $(\lambda x\,.\,\lambda x\,.\,xu)\,(\lambda x\,.\,v)$

Введённым понятиям можно придать простую неформальную интерпретацию. Можно считать, что λ-*абстракция* обозначает функцию одной переменной, причем переменная, указанная после символа λ является ее *формальным* параметром. При таком подходе *применение* MN означает вызов функции M с передачей ей в качестве *фактического* параметра выражения N. То, что при этом используется бесскобочная запись, может напоминать, например, о выражениях с тригонометрическими функциями или линейными операторами:

$$\operatorname{tg} x = \frac{\sin x}{\cos x}.$$

2.2 Структура термов и подстановка

Дадим несколько определений, характеризующих внутреннюю структуру λ-термов.

Определение 2 *Областью действия (scope)* λ-*абстракции* λx *в терме* $(\lambda x\,.\,M)$ *называется терм* M*.*

Определение 3 *Вхождением переменной в терме называется любое ее использование внутри этого терма (в том числе и сразу после символа* λ*).*

Определение 4 *Вхождение переменной* x *в терм* M *называется*

- *связанным (bound), если оно находится в области действия некоторой абстракции* λx *в терме* M*;*
- *связанным и связывающим (binding), если оно как раз и является буквой* x *в абстракции* λx*;*
- *свободным (free) во всех остальных случаях.*

Заметим, что в этом определении идёт речь о связанных и свободных *вхождениях* переменной x, а не о самой переменной x. Переменная, вообще говоря, может входить в терм несколько раз, причем некоторые из ее вхождений могут оказаться свободными, а другие — связанными, например в терме

$$x\,\lambda x\,.\,x$$

есть три вхождения переменной x: первое вхождение является свободным, второе связывающим, а третье — связанным.

Определение 5 *Множеством свободных переменных терма* P*, обозначаемым* $FV(P)$ *(free variables), будем называть множество переменных, имеющих в терме* P *хотя бы одно свободное вхождение:*

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda x.M) & = & FV(M)\backslash\{x\} \\ FV(MN) & = & FV(M) \cup FV(N) \end{array}$$

Следует обратить внимание на то, что структура этого определения повторяет структуру определения множества λ-термов.

Определение 6 *Множеством связанных переменных терма P, обозначаемым $BV(P)$ (bound variables), будем называть множество переменных, имеющих в терме P хотя бы одно связанное вхождение:*

$$\begin{array}{lcl} BV(x) & = & \varnothing \\ BV(\lambda x.M) & = & \{x\} \cup BV(M) \\ BV(MN) & = & BV(M) \cup BV(N) \end{array}$$

В соответствии с этими определениями, множества свободных и связанных переменных терма $x\,\lambda x\,.\,x$ будут совпадать:

$$\begin{array}{ll} FV(x\,\lambda x\,.\,x) & = FV(x) \cup FV(\lambda x\,.\,x) = \{x\} \cup FV(x) = \\ & = \{x\} \cup \{x\} = \{x\}; \\ BV(x\,\lambda x\,.\,x) & = BV(x) \cup BV(\lambda x\,.\,x) = \varnothing \cup (\{x\} \cup BV(x)) = \\ & = \varnothing \cup (\{x\} \cup \varnothing) = \{x\}. \end{array}$$

В вводных курсах по программированию (независимо от используемого языка программирования) обычно используют следующее несколько упрощённое объяснение смысла формальных и фактических параметров: при вызове подпрограммы предварительно вычисленные значения фактических параметров подставляются в теле подпрограммы вместо ее формальных параметров, после чего начинает исполняться собственно подпрограмма. Это соображение можно использовать и в λ-исчислении, однако необходимо точно определить, что именно понимается под *подстановкой*. В том, что это не так просто, как может показаться на первый взгляд, можно убедиться, дав следующее *неправильное* определение, снова отражающее структуру определения множества λ-термов.

Неправильное определение 1 *Операцией подстановки терма N (фактический параметр) в терм P (подпрограмма) вместо переменной x (формальный параметр), обозначаемой*[6] *как $[N/x]P$, называется операция преобразования терма, выполняемая по правилам:*

$$\begin{array}{lcl} [N/x]x & = & N; \\ [N/x]y & = & y, \quad \textit{если } x \neq y; \\ [N/x](PQ) & = & ([N/x]P)([N/x]Q); \\ [N/x](\lambda x\,.\,P) & = & \lambda x\,.\,([N/x]P); \\ [N/x](\lambda y\,.\,P) & = & \lambda y\,.\,([N/x]P), \quad \textit{если } x \neq y. \end{array}$$

Проблемы этого определения скрываются в двух последних правилах. Понятно, что при выполнении подстановки не должен меняться смысл терма (точнее, наше интуитивное представление о его интерпретации), однако

[6] В разных источниках используются абсолютно разные обозначения операции подстановки, поэтому при их чтении следует убедиться, что смысл обозначения понят корректно.

посмотрим, что произойдет с термом $\lambda x\,.\,x$ при проведении подстановки $[t/x]$ по сформулированным выше правилам:

$$[t/x](\lambda x\,.\,x) = \lambda x\,.\,([t/x]x) = \lambda x\,.\,t.$$

Получается, что тождественная функция превратилась в константу, а связанная в исходном терме переменная x исчезла, превратившись в свободную переменную t.

Теперь выполним подстановку $[y/x]$ в терме $\lambda y\,.\,x$:

$$[y/x](\lambda y\,.\,x) = \lambda y\,.\,([y/x]x) = \lambda y\,.\,y.$$

Произошло нечто противоположное: константа превратилась в тождественную функцию. В таких случаях обычно говорят, что произошёл *захват* свободной переменной y, имея в виду, что в подставляемом терме y выступала в качестве свободной переменной, а после подстановки она же оказалась связанной.

Таким образом, корректное определение подстановки должно учитывать связанные и свободные переменные, препятствуя исчезновения одних и захвату других. Проблему с исчезновением связанной переменной можно решить с помощью запрета на распространение подстановки $[N/x]$ внутрь абстракции $\lambda x\,.\,P$, причём это вполне согласуется с пониманием смысла формального параметра подпрограммы: он никак не зависит от действий во внешних по отношению к данной подпрограмме частях программы. Проблема с захватом свободных переменных может быть решена с помощью *переименования* связывающих переменных в соответствующих λ-абстракциях.

Следующее определение полностью отражает предпринятые решения.

Определение 7 *Операцией подстановки терма N в терм P вместо переменной x, обозначаемой как $[N/x]P$, называется операция преобразования терма, выполняемая по правилам:*

$$\begin{array}{lcll}
[N/x]x & = & N & \\
[N/x]y & = & y, & \text{если } x \neq y \\
[N/x](PQ) & = & ([N/x]P)([N/x]Q) & \\
[N/x](\lambda x\,.\,P) & = & \lambda x\,.\,P & \\
[N/x](\lambda y\,.\,P) & = & \lambda y\,.\,P, & \text{если } x \notin FV(P) \\
[N/x](\lambda y\,.\,P) & = & \lambda y\,.\,([N/x]P), & \text{если } x \in FV(P)\ и\ y \notin FV(N) \\
[N/x](\lambda y\,.\,P) & = & \lambda z\,.\,([N/x]([z/y]P)), & \text{если } x \in FV(P)\ и\ y \in FV(N), \\
 & & & \text{причём } z \notin FV(NP)
\end{array}$$

Вместо двух последних правил неправильного определения в правильном появились четыре новых правила, первое из которых решает проблему исчезновения связанных переменных, а три последних — проблему захвата свободных. Протестируем новое определение на старых примерах:

$$[t/x](\lambda x\,.\,x) = \lambda x\,.\,x,$$
$$[y/x](\lambda y\,.\,x) = \lambda z\,.\,([y/x]([z/y]x)) = \lambda z\,.\,([y/x]x) = \lambda z\,.\,y.$$

Действительно, теперь тождественная функция остаётся тождественной, а константная — константной, правда, с точностью до имени связывающей переменной.

Определение 8 *Замена терма $\lambda x\,.\,M$ на терм $\lambda y\,.\,[y/x]M$ (т.е. переименование связанной переменной) называется α-конверсией. Два терма P и Q, один из которых можно получить из другого с помощью конечного числа α-конверсий, называются конгруэнтными, что обозначается как $P \equiv_\alpha Q$.*

В нашей интерпретации конгруэнтные термы можно отождествлять, так как переименование формального параметра подпрограммы с одновременной заменой имени параметра в её теле не приводит к изменению смысла ни данной подпрограммы, ни программы в целом.

Например, следующие термы конгруэнтны:

$$\begin{array}{lll} \lambda x\,.\,x & \equiv_\alpha & \lambda y\,.\,y; \\ y\,\lambda y\,.\,y & \equiv_\alpha & y\,\lambda z\,.\,z; \\ \lambda x\,.\,(\lambda y\,.\,x) & \equiv_\alpha & \lambda y\,.\,(\lambda x\,.\,y). \end{array}$$

Убедиться в конгруэнтности последней пары термов можно, переименовав последовательно x в t, y в x и t в y.

2.3 Редукция термов

Определение 9 *Любой терм вида*

$$(\lambda x\,.\,M)N$$

называется редексом или редуцируемым выражением (reducible expression).

Если считать λ-абстракцию функцией, а применение — вызовом функции, то редекс представляет собой вызов функции, явно заданной абстракцией. Выполнив такой вызов, т.е. подставив фактический параметр на место формального, можно получить результат вызова:

$$[N/x]M.$$

Определение 10 *Отношение, которое ставит в соответствие редексу $(\lambda x\,.\,M)N$ терм $[N/x]M$ называется β-редукцией. Если терм P содержит в качестве своего фрагмента редекс $(\lambda x\,.\,M)N$, который сводится (редуцируется) к терму $[N/x]M$, причем после замены редекса на этот терм получается терм P', то будем писать, что*

$$P \to_\beta P'.$$

Если терм P' можно получить из терма P с помощью конечного числа шагов β-редукции, будем обозначать этот факт как

$$P \to_\beta^* P'.$$

Рассмотрим несколько примеров:

(1) $(\lambda x\,.\,x)\,y \to_\beta y$;

(2) $(\lambda x\,.\,y)\,t \to_\beta y$;

(3) $(\lambda x\,.\,x(xy))N \to_\beta N(Ny)$.

Определение 11 *Говорят, что λ-терм находится в нормальной форме, если он не содержит ни одного редекса.*

Для приведения терма в нормальную форму, вообще говоря, может потребоваться несколько шагов редукции:

$$\begin{array}{ll}(\lambda x\,.\,\lambda y\,.\,yx)\,z\,v & \to_\beta ([z/x](\lambda y\,.\,yx))\,v = (\lambda y\,.\,yz)\,v \\ & \to_\beta [v/y](yz) = vz; \\ (\lambda x\,.\,\lambda y\,.\,x)(\lambda z\,.\,z)\,t & \to_\beta ([\lambda z\,.\,z/x](\lambda y\,.\,x))\,t = (\lambda y\,.\,\lambda z\,.\,z)\,t \\ & \to_\beta [t/y](\lambda z\,.\,z) = \lambda z\,.\,z.\end{array}$$

Следующий пример показывает, что не все термы имеют нормальную форму:

$$\begin{array}{ll}(\lambda x\,.\,xx)(\lambda x\,.\,xx) & \to_\beta [\lambda x\,.\,xx/x](xx) = (\lambda x\,.\,xx)(\lambda x\,.\,xx) \\ & \to_\beta [\lambda x\,.\,xx/x](xx) = (\lambda x\,.\,xx)(\lambda x\,.\,xx) \\ & \to_\beta \ldots\end{array}$$

Более того, некоторые термы после «редуцирования» только увеличиваются, например:

$$\begin{array}{ll}(\lambda x\,.\,xxy)(\lambda x\,.\,xxy) & \to_\beta [\lambda x\,.\,xxy/x](xxy) = (\lambda x\,.\,xxy)(\lambda x\,.\,xxy)y \\ & \to_\beta ([\lambda x\,.\,xxy/x](xxy))y = (\lambda x\,.\,xxy)(\lambda x\,.\,xxy)yy \\ & \to_\beta \ldots\end{array}$$

С помощью этого «расширяющегося» терма можно заметить интересный эффект, рассмотрим терм вида:

$$(\lambda y\,.\,v)((\lambda x\,.\,xxy)(\lambda x\,.\,xxy))$$

Ясно, что с одной стороны (вычисляемый в данный момент редекс подчёркнут):

$$\underline{(\lambda y\,.\,v)((\lambda x\,.\,xxy)(\lambda x\,.\,xxy))} \to_\beta v,$$

а с другой

$$\begin{array}{ll}(\lambda y\,.\,v)(\underline{(\lambda x\,.\,xxy)(\lambda x\,.\,xxy)}) & \to_\beta (\lambda y\,.\,v)(\underline{(\lambda x\,.\,xxy)(\lambda x\,.\,xxy)}y) \\ & \to_\beta (\lambda y\,.\,v)(\underline{(\lambda x\,.\,xxy)(\lambda x\,.\,xxy)}yy) \\ & \to_\beta \ldots\end{array}$$

Таким образом, приход к нормальной форме в результате редукции зависит от порядка шагов редукции: «неправильная» последовательность редукции может никогда не привести к нормальной форме, даже если она существует.

Возникает вопрос: всегда ли в конечном итоге будет получаться одна и та же нормальная форма (при условии, что она вообще будет получаться), т.е. единственна ли она. Ответ на этот вопрос даёт теорема Чёрча—Россера, одна из важнейших теорем λ-исчисления.

2.3.1 Теорема Чёрча—Россера

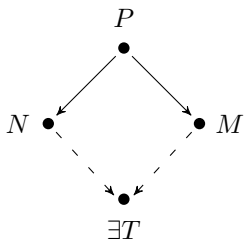
Рассмотрим еще один пример, в котором разный порядок редукции приводит к одинаковым результатам (вычисляемый первым редекс подчёркнут):

$$\begin{array}{ll} (\lambda x\,.\,\underline{(\lambda y\,.\,yx)\,z})\,v & \to_\beta (\lambda x\,.\,[z/y](yx))\,v = (\lambda x\,.\,zx)\,v \\ & \to_\beta [v/x](zx) = zv; \\ \underline{(\lambda x\,.\,(\lambda y\,.\,yx)\,z)\,v} & \to_\beta [v/x]((\lambda y\,.\,yx)\,z) = (\lambda y\,.\,yv)\,z \\ & \to_\beta [z/y](yv) = zv. \end{array}$$

Имеет место следующий важный результат, который обычно называется теоремой Чёрча—Россера.

Теорема 1 *Если $P \to_\beta^* N$ и $P \to_\beta^* M$, то существует такой терм T, что $N \to_\beta^* T$ и $M \to_\beta^* T$.*

Это свойство β-редукции иногда называется свойством Чёрча—Россера или свойством ромба[7], поскольку графически его можно изобразить так:



Пользуясь этой теоремой, можно, наконец, ответить на вопрос о единственности нормальной формы λ-термов.

Следствие 1 *Если терм имеет хотя бы одну нормальную форму, то все его нормальные формы совпадают с точностью до конгруэнтности.*

2.3.2 Стратегии редукции

Так как существование нормальной формы не гарантируется, а к тому же при некоторых порядках редукции нормальная форма может вообще не возникнуть, даже если она существует, необходимо сформулировать правила, позволяющие редуцировать выражения с наибольшей вероятностью успеха. Порядок сведения редексов обычно называют *стратегией редукции*.

[7] По-английски это свойство обозначается термином *diamond property*, поэтому на русский язык его также можно перевести как алмазное или даже бубновое свойство, учитывая разные значения английского слова diamond.

Определение 12 *Стратегия редукции, при которой на каждом шаге выбирается самый левый, самый внешний редекс, называется нормальным порядком редукции.*

Теорема 2 *Нормальный порядок вычислений всегда приводит к нормальной форме, при условии, что она существует.*

Определение 13 *Стратегией редукции с вызовом по имени называется порядок редукции, при котором на каждом шаге выбирается самый левый, самый внешний редекс, причем редукция внутри абстракций не производится.*

Можно также определить модифицированную стратегию с вызовом по имени, которая называется *вызовом по необходимости*. В рамках этой стратегии при первом использовании аргумента его значение (нередуцируемая форма) вычисляется, а все остальные его вхождения заменяются на это значение.

Определение 14 *Стратегией редукции с вызовом по значению называется порядок редукции, при котором на каждом шаге выбираются только внешние редексы, причем их правые части предварительно вычисляются.*

Стратегию с вызовом по значению иногда называют *аппликативной*. Говорят, что стратегия с вызовом по значению является *строгой* в том смысле, что значения аргументов всегда вычисляются независимо от того, используются они в теле функции или нет. При *нестрогих* (или *ленивых*) вычислениях — при вызове по имени или по необходимости — вычисляются только те аргументы, которые действительно используются.

Следует заметить, что все перечисленные стратегии редукции, кроме нормального порядка, не производят редукцию внутри абстракций. Эти стратегии приводят к так называемой *слабой (головной) нормальной форме*, т.е. форме, не содержащей внешних редексов.

3 λ-исчисление как язык программирования

λ-исчисление можно рассматривать как простой язык программирования. Далее нашей задачей будет кодирование базовых элементов любого языка программирования в терминах λ-исчисления. В числе таких элементов будут логические значения и операции, натуральные числа, пары значений, условная операция и рекурсивный вызов.

3.1 Представление данных в λ-исчислении

Для начала введём логические значения $true$ и $false$:

$$true = \lambda x\,.\,\lambda y\,.\,x; \quad false = \lambda x\,.\,\lambda y\,.\,y.$$

Смысл знака $=$ в этих выражениях заключается в следующем: мы *договариваемся* обозначать термы, стоящие в правой части равенства, символами из левой части. Вообще говоря, назвать истиной и ложью можно любые термы, но такой выбор обозначений оказался наиболее удобным. Истинное значение можно трактовать как функцию двух аргументов, возвращающую первый из них, а ложное значение — как возвращающую второй.

Используя эти определения, можно ввести *условное выражение*:

$$\text{if } C \text{ then } E_1 \text{ else } E_2 = CE_1E_2.$$

Действительно, вычислим:

$$\text{if } true \text{ then } E_1 \text{ else } E_2 = true\ E_1E_2 = (\lambda x\,.\,\lambda y\,.\,x)E_1E_2 = E_1.$$

С другой стороны:

$$\text{if } false \text{ then } E_1 \text{ else } E_2 = false\ E_1E_2 = (\lambda x\,.\,\lambda y\,.\,y)E_1E_2 = E_2.$$

Пользуясь условной операцией, нетрудно определить отрицание, конъюнкцию и дизъюнкцию:

$$\begin{array}{lcllcl} \text{not } P & = & \text{if } P \text{ then } false \text{ else } true & = & P\ false\ true; \\ P \text{ and } Q & = & \text{if } P \text{ then } Q \text{ else } false & = & P\ Q\ false; \\ P \text{ or } Q & = & \text{if } P \text{ then } true \text{ else } Q & = & P\ true\ Q. \end{array}$$

Данные могут объединяться в структуры данных, простейшей такой структурой является пара:

$$(E_1, E_2) = \lambda z\,.\,z\ E_1\ E_2.$$

Для работы с парами необходимы функции доступа к отдельным компонентам:

$$\begin{array}{lcl} \text{fst } P & = & P\ true; \\ \text{snd } P & = & P\ false. \end{array}$$

Убедимся в работоспособности этих функций:

$$\begin{array}{lcl} \text{fst } (E_1, E_2) & = & (\lambda z\,.\,z\ E_1\ E_2)\,true = true\ E_1\ E_2 = E_1; \\ \text{snd } (E_1, E_2) & = & (\lambda z\,.\,z\ E_1\ E_2)\,false = false\ E_1\ E_2 = E_2. \end{array}$$

Наконец, натуральные числа можно ввести так:

$$\begin{array}{rcl} 0 & = & \lambda x\,.\,\lambda y\,.\,y \\ 1 & = & \lambda x\,.\,\lambda y\,.\,xy \\ 2 & = & \lambda x\,.\,\lambda y\,.\,x(xy) \\ 3 & = & \lambda x\,.\,\lambda y\,.\,x(x(xy)) \\ & \ldots & \end{array}$$

Натуральные числа, представленные таким образом, называют *нумералами Чёрча*. Нумерал Чёрча можно интерпретировать как функцию двух переменных, которая применяет свой первый аргумент к второму соответствующее число раз, например:

$$\begin{array}{rcl} 2\ f\ t & = & (\lambda x\,.\,\lambda y\,.\,x(xy))\,f\ t \\ & = & (\lambda y\,.\,f(fy))\,t \\ & = & f(ft). \end{array}$$

Можно ввести *функцию следования*, которая ставит в соответствие каждому натуральному числу следующее за ним ($n \mapsto n+1$):

$$\text{succ} = \lambda n\,.\,\lambda x\,.\,\lambda y\,.\,x(nxy).$$

Вычислим число, следующее за числом 3:

$$\begin{array}{rcl} \text{succ } 3 & = & (\lambda n\,.\,\lambda x\,.\,\lambda y\,.\,x(nxy))(\lambda x\,.\,\lambda y\,.\,x(x(xy))) \\ & = & \lambda x\,.\,\lambda y\,.\,x\,((\lambda x\,.\,\lambda y\,.\,x(x(xy)))xy) \\ & = & \lambda x\,.\,\lambda y\,.\,x\,((\lambda y\,.\,x(x(xy)))y) \\ & = & \lambda x\,.\,\lambda y\,.\,x\,(x(x(xy))) \\ & = & 4. \end{array}$$

Заметим, что такое определение соответствует нашей интерпретации: терм, находящийся в скобках (nxy) означает n-кратное применение x к y, после этого следует еще одно применение x к полученному ранее результату. В итоге получается, что x применяется к y в точности $n+1$ раз. В целом, процесс вычислений можно трактовать как $(n+1)$-кратное прибавление единицы к нулю. Пользуясь той же интуицией, можно сформулировать и другое определение функции следования, прибавляя n раз единицу к единице:

$$\text{succ}' = \lambda n\,.\,\lambda x\,.\,\lambda y\,.\,nx(xy).$$

Связь между натуральными числами и логическими значениями можно организовать, определив функцию, проверяющую, является ли заданное число нулём:

$$\text{is\_zero?} = \lambda n\,.\,n\,(\lambda x\,.\,false)\,true.$$

Действительно:

$$\begin{array}{rcl}
\text{is_zero? } 0 & = & (\lambda n\,.\,n\,(\lambda x\,.\,false)\,true)\,(\lambda x\,.\,\lambda y\,.\,y) \\
& = & (\lambda x\,.\,\lambda y\,.\,y)(\lambda x\,.\,false)\,true \\
& = & (\lambda y\,.\,y)\,true \\
& = & true; \\
\text{is_zero? } 1 & = & (\lambda n\,.\,n\,(\lambda x\,.\,false)\,true)\,(\lambda x\,.\,\lambda y\,.\,xy) \\
& = & (\lambda x\,.\,\lambda y\,.\,xy)(\lambda x\,.\,false)\,true \\
& = & (\lambda y\,.\,(\lambda x\,.\,false)\,y)\,true \\
& = & (\lambda x\,.\,false)\,true \\
& = & false.
\end{array}$$

Аналогично, для всех ненулевых n функция $is\_zero?$ возвращает $false$.

Функции сложения и умножения чисел можно определить следующим образом:

$$\begin{array}{rcl}
m+n & = & \lambda x\,.\,\lambda y\,.\,mx(nxy); \\
m*n & = & \lambda x\,.\,\lambda y\,.\,m(nx)y.
\end{array}$$

Возможность проверить справедливость этих определений на примерах предоставляется читателю, но ясно, что интуитивная интерпретация сохраняется: чтобы сложить m и n, нужно n раз прибавить единицу к нулю (применить x к y), а затем прибавить к полученному результату единицу еще m раз; чтобы умножить m на n, достаточно m раз прибавить n единиц к нулю.

Интересно, что *функция предшествования* pred и операция вычитания определяются гораздо более сложным образом. Идея этого определения принадлежит ученику Чёрча Стивену Клини (1909—1994)[8]. Суть его идеи заключается в использовании вспомогательной функции, которая любой паре вида (m, n), где m и n — нумералы Чёрча, ставит в соответствие пару $(n, n+1)$. Определить такую функцию нетрудно:

$$\text{step} = \lambda p\,.\,(\text{snd } p, \text{succ}(\text{snd } p)).$$

Здесь активно используется введённое ранее обозначение пары, функции доступа к второму элементу пары и функции следования, однако понятно, что определение можно записать и непосредственно в виде λ-терма.

Теперь, для того чтобы найти число, предшествующее заданному m достаточно m раз применить функцию step к паре $(0, 0)$, после этого первый компонент получившейся пары будет равен в точности $m - 1$:

$$\text{pred} = \lambda m\,.\,\text{fst}(m \text{ step } (0, 0)).$$

Будучи применённой к нулю, функция pred даёт ноль. Пользуясь этой функцией, можно определить вычитание:

$$m - n = n \text{ pred } m.$$

[8] Определение функции предшествования занимало мысли Чёрча в течение нескольких месяцев, над этой же задачей размышлял и его ученик. Сложно удержаться от приятных мыслей, представляя ситуацию, в которой молодой Клини вбегает в кабинет своего учителя с криками: „Я знаю, как вычесть единицу!“

3.2 Комбинаторы неподвижной точки и рекурсия

Очередной шаг — введение такого важного элемента языка программирования, как рекурсия. Однако здесь есть некоторая сложность: обычно для вызова функции внутри её тела используется её же имя, но λ-абстракции имени не имеют. Тем не менее, организовать рекурсивные вызовы возможно. Это обеспечивается существованием *комбинаторов неподвижной точки.*

Определение 15 *Терм называется замкнутым (или комбинатором), если он не содержит ни одной свободной переменной.*

Определение 16 *Комбинатором неподвижной точки называется такой замкнутый терм Y, для которого при любом f выполняется равенство:*

$$f(Yf) = Yf.$$

Название комбинатора объясняется тем, что он находит такую точку из области определения функции f, которая переводится этой функцией сама в себя, т.е. является неподвижной.

Существует достаточно большое количество комбинаторов неподвижной точки, одним из них является комбинатор Тьюринга:

$$Y_T = AA, \text{ где } A = \lambda x\,.\,\lambda y\,.\,y(xxy).$$

Проверим, что комбинатор Тьюринга действительно является комбинатором неподвижной точки:

$$\begin{array}{rcl} Y_T f & = & AAf \\ & = & (\lambda x\,.\,\lambda y\,.\,y(xxy))Af \\ & = & (\lambda y\,.\,y(AAy))f \\ & = & f(AAf) \\ & = & f(Y_T f). \end{array}$$

Теперь, следуя давней традиции, реализуем рекурсивную функцию вычисления факториала числа. Функцию fact хотелось бы определить следующим образом:

$$\text{fact } n = \text{if is\_zero? } n \text{ then 1 else } n * fact(\text{pred } n)$$

Или, что то же самое, как

$$\text{fact} = \lambda n\,.\,\text{if is\_zero? } n \text{ then 1 else } n * \text{fact}(\text{pred } n)$$

Очевидная проблема этого определения — присутствие fact в теле функции. Чтобы его избежать, определим факториал как функцию от функции:

$$\text{fact} = (\lambda f\,.\,\lambda n\,.\,\text{if is\_zero? } n \text{ then 1 else } n * f(\text{pred } n))fact$$

Обозначим выражение в скобках через H:

$$H = \lambda f\,.\,\lambda n\,.\,\text{if is\_zero? } n \text{ then 1 else } n * f(\text{pred } n)$$

Ясно, что H — это вполне законный λ-терм, а именно, λ-абстракция. Теперь видно, что

$$\text{fact} = H\ \text{fact}$$

Последнее равенство означает, что функция fact является неподвижной точкой абстракции H, а значит, её можно найти с помощью комбинатора неподвижной точки[9]:

$$\text{fact} = Y_T H.$$

3.3 Локальные объявления

В принципе, полный по Тьюрингу язык программирования на основе λ-исчисления построен. Любая программа на этом языке — это λ-абстракция. Запуск программы означает, что этой абстракции передается в качестве параметра исходное значение. Результат редукции — результат выполнения программы. Таким результатом оказывается некоторое выражение в слабой нормальной форме. Чтобы программирование на этом языке стало еще более удобным, добавим возможность создания локальных объявлений, т.е. способ именования вспомогательных конструкций.

Введём в рассмотрение следующее обозначение:

$$(\text{let } x = s \text{ in } t) = (\lambda x\,.\,t)\,s$$

и приведём простой пример его использования:

$$\begin{array}{rcl}(\text{let } z = 2+3 \text{ in } z*z) & = & (\lambda z\,.\,z*z)\,(2+3) \\ & = & (2+3)*(2+3)\end{array}$$

Конструкцию let/in можно использовать последовательно:

$$\text{let } x = 1 \text{ in } (\text{let } x = 2 \text{ in } (\text{let } y = x \text{ in } x+y)) = 4$$

Можно также определить параллельное объявление нескольких имён:

$$\begin{array}{l}\text{let} \\ \quad x_1 = s_1 \\ \quad x_2 = s_2 \\ \quad \ldots \\ \quad x_n = s_n \\ \text{in } t = (\lambda x_1\,.\,\lambda x_2\,.\,\ldots\,\lambda x_n\,.\,t)\,s_1 s_2 \ldots s_n\end{array}$$

Приведём пример его использования:

$$\begin{array}{l}\text{let } x = 1 \text{ in} \\ \qquad\qquad \begin{array}{l}\text{let} \\ \quad x = 2 \\ \quad y = x \\ \text{in } x + y = 3\end{array}\end{array}$$

[9] Здесь мы неявно пользуемся тем фактом, что неподвижная точка является единственной, и комбинатор Тьюринга вычислит именно её. Вообще говоря, соответствующий факт требует доказательства, он выводится в рамках теории порядка и составляет содержание известной теоремы Клини о неподвижной точке.