

Haskell — специфика языка (продолжение)

Карринг и лямбда-абстракция

Карринг (currying). Отметим, что в Haskell функции $f'(x, y) = x+y$ и $f' ' x y = x+y$ — разные по сути функции. Если $f'(x, y)$ — это функция от одной переменной (кортежа или вектора (x, y)), то $f' ' x y$ — это функция от двух переменных. Тип у них тоже будет разный, и это можно будет узнать командой `:t`.

Для чего нужен карринг и такие сложности? С теоретической точки зрения, например, карринг позволяет сводить рассмотрение функций от многих переменных к функциям одной переменной.

[Currying on wiki.haskell.org](https://wiki.haskell.org/Currying)

[Немного о каррировании в Haskell on habr.com](https://habr.com/ru/post/444444/)

[Higher order function on wiki.haskell.org](https://wiki.haskell.org/Higher_order_function)

Именно так и реализованы функции в Haskell! За счет механизма каррирования все функции на самом деле одноместные.

Рассмотрим вновь выше заданную функцию $f' ' x y$, назовем ее более логично `add x y`:

`add x y = x + y`

С точки зрения Haskell, при применении функции к своим аргументам мы на самом деле получаем следующее:

`add x y = (add x) y`

соответственно и тип функции в реальности вместо, например,

`add :: Integer -> Integer -> Integer`

будет

`add :: Integer -> (Integer -> Integer)`

таким образом, промежуточная функция `(add x)` называется *частичным применением*, и с «замороженным аргументом» `x` имеет тип:

`(add x) :: Integer -> Integer`

Получили семейство функций, где переменная `x` выступает в роли параметра, позволяющего задавать ту или иную функцию от одного переменного (его играл роль `y`).

Отметим, что самой по себе, формально, такой функции как `add x` выше у нас быть не может и интерпретатор даст ошибку:

```
ghci> :t (add x)
<interactive>:1:6: error: Variable not in scope: x :: Int
```

И если хотим проверить тип, то следует использовать на месте `x` чисел (констант), хотя и произвольных:

```
ghci> :t add 2
add 2 :: Integer -> Integer
```

Поэтому и говорилось о «замороженном аргументе» x ...

Далее, для большей ясности стоит рассмотреть пример, но с аргументами разного типа:

```
sample :: Int -> Char -> Bool
sample n c = (n < 3) && (c == 'a')
ghci> sample 3 'q'
False
ghci> sample 2 'a'
True
ghci> :t sample 1
sample 1 :: Char -> Bool
```

Тем не менее, записи вида `add x` или `sample n` могут встречаться в правых частях определений других функций:

```
f x = add x
```

при этом функция `f` будет от двух аргументов, как и функция `add`!

[Partial application on wiki.haskell.org](http://wiki.haskell.org/Partial_application)

[Функциональное программирование на Haskell. Ч.2. Основные типы и классы \(ibm developerworks\)](#)

[Функциональные типы и композиция функций в Хаскелле](#)

Но вернёмся к каррированию и его использованию. На практике карринг позволяет задавать функции неполным применением.

Рассмотрим вновь `add x y`, теперь мы легко можем задать частный случай сложения, функцию-инкремент `add1 t = 1+t`, просто частичным применением уже заданной функции `add` с константой (и это более осмысленный случай, чем использование `add x` с переменной):

```
add1 = add 1
```

Переход от каррированной к «традиционной» версии функции и наоборот (в случае двух переменных), можно осуществлять с помощью специальных функций `curry` и `uncurry`. Например,

```
p1s = uncurry (+)
> p1s (2,3)
> 5
```

```
p1 = curry p1s
> p1 2 3
> 5
```

Лямбда-абстракции. Хотя идея лямбда-исчисления уходит корнями в фундаментальное исчисление, придуманное Чёрчем и реализованное потом МакКарти в Lisp — данный синтаксис прежде всего предназначен для описания анонимных функций и случаев, аналогичных частичному применению (функций). Однако полезен и для обычного описания функций.

```
mi2 = \x -> (x-2)
```

```
trio :: Integer -> Integer -> Integer -> Integer
trio = \x y z -> x*y + z
```

Примеры частичного применения функции ниже показывают, что если мы даем частичное определение функции без последнего аргумента (каррированной функции), то получается естественное частичное применение. Если мы хотим оставить в «свободном плавании» первый или второй аргумент, то мы вынуждены использовать лямбда-абстракцию.

```
duo'  = trio 2 3
duo'' = \t -> trio 2 t 3
```

Если мы подставим в качестве аргумента 4, то увидим разницу значений функций duo' и duo''.

В следующем, более простом примере, функции padd1 и padd3 эквивалентны, хотя заданы различными способами (можно запустить функции для проверки с аргументами 4 и 5):

```
padd      = \x y -> x^2 + y
padd1 t = \x -> padd t x
padd2 t = \x -> padd x t
padd3 x = padd x
```

Необходимость в анонимных функциях возникает, например, при использовании функций высокого порядка (т.е. таких, которые в качестве аргумента сами получают функции). Например, рассмотрим функцию **map**, которая применяет полученную в качестве аргумента функцию к списку, полученному в качестве второго аргумента. Так, мы могли бы передать этой функции уже готовую функцию, а могли бы задать ее «на лету». Для просмотра результатов в ghci наберите **show l1** и **show l2**.

```
l1 = map abs [(-1), 2, (-4.2)]
l2 = map (\x -> if x >= 0
               then x else negate x)
      [(-1), 2, (-4.2)]
```

Сигнатура при использовании лямбда-абстракций (особенно, когда у вводимой функции нет имени) может быть определена в скобках по месту использования:

```
l3 =
  map ((\x -> if x >= 0
               then x
               else negate x)::(Double -> Double))
      [(-1), 2, (-4.2)]
```

<https://ru.wikipedia.org/wiki/%D0%9B%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5>

Замыкания (локальные определения)

Дальними аналогами локальных присваиваний в императивных языках в функциональном языке Haskell являются два способа порождать локальные определения.

let-выражения.

```
roots a b c =  
  let d = b^2 - 4*a*c  
      sd = sqrt d  
      x1 = (-b - sd) / (2*a)  
      x2 = (-b + sd) / (2*a)  
  in (x1,x2)
```

where-конструкция. Данная конструкция не является выражением, она является частью синтаксиса объявления функций и **case**-выражений. Хотя в применении они очень похожи!

```
roots' a b c = (x1,x2) where  
  d = b^2 - 4*a*c  
  sd = sqrt d  
  x1 = (-b - sd) / (2*a)  
  x2 = (-b + sd) / (2*a)
```

Пример вычисления корней квадратного уравнения с ветвлением

При упрощённом подходе мы не отслеживаем случаи, когда старший коэффициент будет равен нулю или дискриминант будет отрицательный. Однако в некоторых случаях это будет приводить к аварийному останову программы в процессе вычисления. Мы можем эти случаи отследить заранее и реагировать, например, более осмысленными предупреждениями:

```
roots' a b c | ((a == 0) && (b == 0)) = error "No roots at all!"  
             | ((a == 0) && (b /= 0)) = (x,x)  
             | (a /= 0) = (x1,x2)  
where  
  d = b^2 - 4*a*c  
  sd = if (d<0)  
        then error ("No real roots! d=" ++ show d)  
        else sqrt d  
  x1 = (-b - sd) / (2*a)  
  x2 = (-b + sd) / (2*a)  
  x = -c/b
```

Операторы композиции функции и применения в Haskell

Два странных (для программистов из «другого» мира) оператора «.» и «\$». Сначала рекомендуется о них прочесть статьи:

[Функциональные типы и композиция функций в Хаскелле](#)

[Еще Одно Руководство по Монадам \(часть 1: основы\)](#)

Понимание их удобства приходит с опытом. Например, оператор композиции позволяет использовать так называемую *бесточечную нотацию* (*бесточечный стиль*), т.е. при определении сложной функции позволяет обходиться без указания аргументов.

Ирония в том, что в «бесточечном стиле» оператор «точка» (.) очень даже используется, — сильнее, чем в обычном коде. Тут правильнее было бы сказать «безаргументный стиль», а не «бесточечный», так как мы опускаем аргументы функций. *Mike Vanier*

Вот как это можно использовать:

```
f x = cos (sin x)
g x = cos $ sin x
h = cos . sin
```

Здесь функции `f`, `g` и `h` задают одну и ту же композицию двух функций `sin` и `cos`.

Практичность оператора применения еще нагляднее: он позволяет не писать лишних скобок, сохраняя возможность работы с переменной.

Отметим, что в математике композиция функций часто определяется в другом порядке:

$$(f \circ g)(x) = g f(x)$$

и это, кстати, хорошо соответствует конвейеру в Linux.

Поэтому, мы можем задать свои операции применения и композиции с обратным (на самом деле, *прямым*) порядком:

```
x $> f = f x
infixl 0 $>
```

```
f .> g = g . f
infixl 9 .>
```

И использовать взаимозаменяемо:

```
*Main> 3 $> (^2) .> (^3)
729
*Main> (^3) . (^2) $ 3
729
```

Операторы в Haskell

Операторы в Haskell — это те же функции, только двухместные, как правило задаваемые символами, и используемые внутри выражений (инфиксно), с различным приоритетом и ассоциативностью. Самый известный пример: операторы сложения, вычитания, умножения:

```
x + y
x - y
x * y
```

[Gentle intro: 3.2 Инфиксные операторы](#)

[ohaskell.guide: Мир операторов](#)

[Операторы в языке программирования Haskell](#)

[Haskell как первый язык программирования. Ч.2](#)

[Operator Glossary](#)

[The Haskell-98 Report \(rus\)](#) в разделе об объявлениях деклараций сообщает следующее:

`infix`-объявление задает ассоциативность и приоритет (силу связывания) одного или более операторов. Целое число `integer` в `infix`-объявлении должно быть в диапазоне от 0 до 9. `infix`-объявление можно разместить всюду, где можно разместить сигнатуру типа. Как и сигнатура типа, `infix`-объявление задает свойства конкретного оператора. Так же, как и сигнатура типа, `infix`-объявление можно разместить только в той же последовательности объявлений, что и объявление самого оператора, и для любого оператора можно задать не более одного `infix`-объявления. (Методы класса являются небольшим исключением: их `infix`-объявления можно размещать в самом объявлении класса или на верхнем уровне.)

По способу ассоциативности операторы делятся на три вида: неассоциативные, левоассоциативные и правоассоциативные (**`infix`**, **`infixl`** и **`infixr`** соответственно). По приоритету (силе связывания) операторы делятся на десять групп, в соответствии с уровнем приоритета от 0 до 9 включительно (уровень 0 связывает операнды наименее сильно, а уровень 9 — наиболее сильно). Если целое число `integer` не указано, оператору присваивается уровень приоритета 9. Любой оператор, для которого нет **`infix`**-объявления, считается объявленным **`infixl 9`**.

Приоритет	Левоассоциативные операторы	Неассоциативные операторы	Правоассоциативные операторы
9	<code>!!</code>		<code>.</code>
8			<code>^</code> , <code>^^</code> , <code>**</code>
7	<code>*</code> , <code>/</code> , <code>`div`</code> , <code>`mod`</code> , <code>`rem`</code> , <code>`quot`</code>		
6	<code>+</code> , <code>-</code>		
5			<code>:</code> , <code>++</code>
4		<code>==</code> , <code>/=</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>`elem`</code> , <code>`notElem`</code>	
3			<code>&&</code>
2			<code> </code>
1	<code>>></code> , <code>>>=</code>		
0			<code>\$</code> , <code>\$!</code> , <code>`seq`</code>

Рис. 1: Встроенные операторы

Операторы можно задавать самим из незанятых комбинаций символов:

```
x >< y = (x,y)
```

```
x <> y = (y,x)
infixr 8 <>
```

и в результате:

```
*Main> 1 >< 2 >< 3
((1,2),3)
*Main> 1 <> 2 <> 3
((3,2),1)
*Main> 1 <> 2 >< 3
((2,3),1)
*Main> 1 >< 2 <> 3
(3,(1,2))
```

И если не определять ассоциативность:

```
x >#< y = (x,y)
infix 8 >#<
```

```
*Main> 1 >#< 2 >#< 3
```

```
<interactive>:2:1: error:
  Precedence parsing error
    cannot mix `>#<' [infix 8] and `>#<' [infix 8] in the same infix expression
```

Кроме того, операторы можно использовать (и определять) как обычные функции, беря их в круглые скобки:

```
*Main> (><) 1 2
(1,2)
```

И, наоборот, обычные функции мы можем использовать как операторы:

```
*Main> plus = (+)
*Main> 1 `plus` 2
3
```

и ещё:

Если ассоциативность и приоритет для `op` не заданы, то по умолчанию используется наивысший приоритет и левоассоциативность. [Переменные, конструкции, операторы и литералы](#)

Postfix

Не очень удобный, на самом деле, в Haskell инструмент, но вот пример его объявления и использования:

```
{-# LANGUAGE PostfixOperators #-}
```

```
-- наличие прагмы
```

```
-- определять обязаны префиксно или инфиксно,  
-- т.е., как обычно
```

```
(!?) 3 = 0  
(!?) _ = 8
```

использование в круглых скобках:

```
> (3 !?)  
0
```

[Alexander Altman. Basic Syntax Extensions](#)

[Postfix operators](#)

Сечения

Частичное применение бинарных операторов называют *сечение*. Например,

```
add1 = (1+)
```

[Haskell-98. 3.5 Сечения](#)

[Section of an infix operator on wiki.haskell.org](#)

Их использование удобно, например, в `map`:

```
map (1+) [34,56,37,40]
```

Это ещё один инструмент — как частичное применение и лямбда-абстракции для создания анонимных функций по месту.

[Операторы в языке программирования Haskell](#)

Использование монад

Пока только пример, для выработки будущей привычки:

```
import Control.Monad.State
```

```
fact' :: Integer -> State Integer Integer
```

```
-- mun состояния - Integer, mun результата - тоже Integer
```

```
fact' 0 = do
```

```
    acc <- get -- получаем накопленный результат
```

```
    return acc -- возвращаем его
```

```
fact' n = do
```

```
    acc <- get -- получаем аккумулятор
```

```
    put (acc * n) -- домножаем его на n и сохраняем
```

```
    fact' (n - 1) -- продолжаем вычисление факториала
```

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

```
fact n = fst $ runState (fact' n) 1
```

```
-- начальное значение состояния = 1
```


где термином *начальное состояние*, *накопленный результат* и *аккумулятор* означаем одно и то же.

Если рассматривать операции `get` и `put` как обращение к некоторому внешнему источнику, то мы видим фактически императивный стиль программирования при определении чистой функции.