Безымянные функции

Поскольку Haskell позволяет работать с функциями также, как и с объектами других типов данных (передавать их в другие функции в качестве аргументов, возвращать из функций в качестве результата и т.п.), время от времени требуется передать в качестве аргумента функцию, которая нигде больше не используется.

Допустим, мы хотим вычислять производную `x*x+2*x-3` в различных точках. Можно, определить функцию `f` и использовать её при вычислении проивзодной в точке:

```
f x =  x*x+2*x-3
diff f 3
diff f 2
…
```

Поскольку в данном случае функция `diff` постоянно вызывается с одним и тем же аргументом, есть смысл определить новую функцию, которая принимает только точку:

```
diffF x = diff f x
```

И тогда вычислять значения, просто обращаясь к ней:

```
diffF 3
diffF 2
```

В данном случае можно заметить, что функция `f` не используется нигде, кроме как в аргументе функции `diff`, и, кроме того, она не использует сопоставления с образцом и состоит всего из одной строки. В таком случае её можно записать прямо в точке вызова, используя конструкцию для определения безымянной функции:

```
diffF x = diff (\x -> x*x+2*x-3) x
```

Поскольку далее будет рассмотрено некоторое число функций, которые принимают в качестве аргументов другие функции, то этот подход может оказаться весьма полезным.

Теперь вспомним про то, что функция в Haskell является значением некоторого функционального типа (подобно тому, как целое число является значением типа `Int`), и то, что константы могут быть объявлены простым указанием их значения:

```
pi :: Float
pi = 3.14
```

Используя безымянные функции мы можем определить некоторые функции таким образом:

```
f :: Float -> Float
f = (\x -> x*x+2*x-3)
```

Это определение отличается от классического только формой записи (кстати, менее наглядной), но имеет тот же смысл – определение функции. Определять именованные функции таким образом достаточно бессмысленно, однако такая запись будет необходима для иллюстрации того, какие функции получаются в итоге вычисления тех или иных выражений.

Частичная применимость

Теперь стоит рассмотреть вторую важную особенность функций в языке Haskell, о которой уже упоминалось выше – ***частичную применимость***. Как уже говорилось в начале, функция от N аргументов с точки зрения Haskell – это то же самое, что функция одного аргумента, которая возвращает остаточную функцию от N-1 аргумента. Рассмотрим функцию сложения:

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

Эта функция имеет два аргумента, однако мы можем применить её к одному аргументу и получить функцию одного аргумента. Тело этой новой функции может быть получено подстановкой первого аргумента в тело исходной функции `add`:

```
(add 1)  ≡ (\y -> 1+y)
```

Затем, применив остаточную функцию ко второму аргументу, мы получим ровно то же самое, как если бы сразу применили функцию `add` к двум аргументам:

```
((add 1) 2) ≡ 3 ≡ (add 1 2)
```

Более того, применение функции к нескольким аргументам (запись справа) воспринимается интерпретатором именно как последовательность применений функций одного аргумента (запись слева). Причем, скобки расставляются именно слева направо, как показано в этой записи.

В свете этого должны стать более понятными синтаксические особенности Haskell, связанные с вызовом функций, такие как:

- Необходимость заключать аргументы в скобки, если они представлены сложными выражениями.
- Отсутствие скобок вокруг аргументов функции.

Определим функцию `inc`, возвращающую увеличенный на единицу аргумент и использующую для этого функцию `add`. Обычный способ это сделать:

```
inc Int -> Int
inc x = add 1 x
```

Если же вспомнить про частичную применимость, то описание этой функции можно задать так:

```
inc :: Int -> Int
inc = add 1
```

Действительно, `inc` мы можем рассматривать как значение функционального типа, которое может быть получено применением функции `add` к одному аргументу. И если при определении такая запись только лишь короче на два символа, то как она меняется, если необходимо продифференцировать функцию `inc` в точке? Разумеется, при условии что такая функция не определена и определять её только ради производной не очень хочется. Используя безымянные функции можно записать так:

```
diff (\x -> add 1 x) 2
```

Однако, есть более короткий вариант:

```
diff (add 1) 2
```

Таким образом, результат применения функции `add` к одному аргументу можно передать в

другую функцию в качестве функционального аргумента (как функцию одной переменной).

Частичное примененение операторов

Очень удобной является синтаксическое сокращение, используемое для частичного применения бинарных операторов. Например, функция инкремента может быть представлена как результат частичного применения оператора `+` к аргументу `1`. Если мы хотим передать ее в функцию дифференцирования, то мы, конечно можем написать что-то вроде:

```
diff (\x -> x + 1) 2
```

Однако, есть значительно более короткая запись:

```
diff (+1) 2
```

Аналогично, `(2*)` - это то же самое, что и `(\x -> 2*x)`. То есть, в отличии от обычных функций при частичном применении операторов можно указывать как первый, так и второй аргументы, что достаточно важно. Действительно, `(/2)` и `(2/)` имеют совершенно разный смысл.

Кортежи

До этого мы рассматривали базовые типы данных. Понятно, что часто возникает необходимость использовать более сложные структуры данных, чем одиночные числа. Простейшим механизмом построения таких более сложных структур в Haskell являются кортежи.

Кортеж – это упорядоченный набор значений фиксированных типов, имеющий фиксированную длину. Множество значений кортежа представляет собой декартово произведение множеств значений компонентов. Рассмотрим примеры значений и типов кортежей:

- `(1,3)` имеет тип `(Int,Int)` - пара целых чисел и может использоваться, например, для представления рациональных чисел;
- `(1.0,2.5,5.3)` имеет тип `(Float,Float,Float)` – тройка чисел с плавающей точкой – например, вектор или точка в трехмерном пространстве;
- `(2,'a')` – `(Int,Char)` – пара из целого числа и символа;
- `('a',2.0,2,6)` – `(Char,Float,Int,Int)` – четверка из символа, дробного числа и двух целых.

Разумеется, компонентами кортежей могут быть и другие кортежи, например:

- `(2,(3,4,5))` – `(Int,(Int,Int,Int))`
- `((1.0,0.0),(0.0,1.0))` – `((Float,Float),(Float,Float))`

Как записывать типы и значения понятно из примеров. Однако, чтобы работать с кортежами необходимо извлекать из них значения компонент. Например, чтобы написать функцию сложения двух векторов необходимо извлечь их координаты и сформировать вектор из соотвтствующих сумм. Для двухкомпонентных кортежей в стандартной библиотеке Haskell определены функции `fst` (сокращение от английского first – первый) и `snd` (сокращение от second), которые позволяют получить соответственно первое и второе значения. Тогда функцию сложения двух двумерных векторов можно написать следующим образом:

```
addVec :: (Float,Float) -> (Float,Float) -> (Float,Float)
```

```
addVec a b = (fst a + fst b,snd a + snd b)
```

Однако, если необходимо складывать вектора большой размерности, то такой подход уже не работает. В данном случае необходимо использовать возможность разбить вектор на компоненты, используя сопоставление с образцом:

```
addVec3 (ax,ay,az) (bx,by,bz) = (ax+bx,ay+by,az+bz)
```

Вообще говоря, такой способ является и более наглядным.

Кроме представления векторов и других сложных структур данных кортежи можно использовать для возврата из функции нескольких значений. Например, функция, которая находит корни квадратного уравнения, может выглядеть следующим образом:

```
roots :: Float -> Float -> Float -> (Float,Float)
roots a b c = ((-b-sd)/(2*a),(-b+sd)/(2*a))
                    where sd = sqrt (b*b - 4*a*c)
```

Синонимы типов

При работе с составными структурами данных обычно принято давать этим структурам имена, отражающие их семантику. В Haskell это может быть сделано с помощью директивы `type`, которая позволяет определить синоним типа:

```
type Point2D = (Float,Float)
```

Тогда, используя это определение, функцию вычисления расстояния между двумя точками можно описать следующим образом:

```
distance :: Point2D -> Point2D -> Float
distance (x1,y1) (x2,y2) = sqrt(dx*dx+dy*dy) where dx = x1-x2
                                                   dy = y1-y2
```

Типы данных

Разумеется, кортежами множество более сложных типов данных не ограничивается и здесь мы рассмотрим использование директивы `data` для определения новых типов. Для начала рассмотрим простой перечислимый тип:

```
data Tribool = Yes | No | Unknown
```

Эта директива определяет тип, имеющий всего три значения. Для того, чтобы понять, как с ним работать, рассмотрим определение следующей функции:

```
tand :: Tribool -> Tribool -> Tribool
tand Yes Yes = Yes
tand No _ = No
tand _ No = No
tand _ _ = Unknown
```

Видно, что значения типа записываются ровно также, как и при его описании, и могут быть использованы при сопоставлении с образцом.

Относительно введённого перечислимого типа стоит отметить, что это “настоящий” перечислимый тип, а не набор констант типа Int, как например в C. Соответственно, никаких функций преобразования в другие типы не задано, их необходимо определять самим.

Однако конструкция `data` позволяет определять не только перечислимые типы. Предположим, что в какой-то структуре необходимо хранить либо число, либо символ – в зависимости от какой-то ситуации. Поскольку типизация в Haskell не позволяет описать функцию, которая принимает число или символ, необходимо описать новый тип, множество значений которого состоит из двух частей – значения, представляющего числа, и значения, представляющего символы. С помощью конструкции `data` это определяется следующим образом:

```
data CharOrInt = CharT Char | IntT Int
```

Очень похоже на перечислимый тип, однако здесь вертикальной чертой разделены не единичные значения, а множества значений. Как создавать значения этого типа:

```
CharT 'a'
IntT 1234
```

То есть, `CharT` и `IntT` являются функциями, которые принимают аргументы типов `Char` и `Int` соответственно и возвращают значение типа `CharOrInt`. Эти функции называются ***конструкторами*** в силу того, что они позволяют конструировать значения типа.

Здесь стоит отметить схожесть определения такого типа с определением перечислимого типа. Действительно, перечислимый тип является всего лишь частным случаем, когда все конструкторы являются функциями нулевой арности (не принимают аргументов).

Кроме конструирования значений, для работы с типом необходимо уметь извлекать из него значения. Для этого используется сопоставление с образцом, аналогично тому, как оно использовалось для перечислимых типов, только с возможностью получить значение, переданное в конструктор. Например, рассмотрим функцию, которая принимает значение типа `CharOrInt` и делает следующее: если это число, то увеличивает его на 1, а если символ, то оставляет неизменным:

```
incInt :: CharOrInt -> CharorInt
incInt (IntT x) = IntT (x+1)
incInt v = v
```

Первая строка определения срабатывает только в том случае, если аргумент сконструирован с помощью `IntT` и извлекает аргумент конструктора в переменную x. Если же аргумент не хранит число, то срабатывает вторая строка, которая возвращает в качестве значения свой аргумент.

В качестве ещё одного примера рассмотрим функцию, которая заменяет символ `'0'` на число ноль:

```
replaceZero :: CharOrInt -> CharorInt
replaceZero (CharT '0') = IntT 0
replaceZero v = v
```

И ещё один пример – функция возводит число в квадрат, если оно больше нуля:

```
squarePos :: CharOrInt -> CharorInt
```

```
squarePos v@(IntT x) | x > 0 = IntT (x*x)
                     | otherwise = v
squarePos v = v
```

В этом примере стоит обратить внимание на образец в первой строке – он состоит из имени переменной, за которой следует символ коммерческого эт, а затем образец, извлекающий содержимое. Это позволяет в правой части использовать как значение, переданное в конструктор аргумента (`x`), так и само значение аргумента (`v`).

Деревья

Теперь перейдём к ещё более сложным структурам данных, которые могут быть определены директивой `data`. Рассмотрим пример, демонстрирующий, как можно представить бинарные деревья, в узлах которых находятся числа. Пустые деревья мы рассматривать не будем, и поэтому выделим два основных варианта описания, каким может быть дерево:

1. Дерево может состоять из одно узла (листа), который хранит число;
2. Дерево может состоять из узла, который хранит число, и имеет два поддерева (каждое из которых является деревом).

В соответствии с этими вариантами опишем дерево:

```
data BiTree = Leaf Int | BiTree Int BiTree BiTree
```

Здесь стоит обратить внимание на три момента:

- Имя одного из конструкторов совпадает с имеем типа – такое допустимо;
- Конструктор может иметь больше одного аргумента;
- Аргументы конструктора могут иметь тип, который мы определяем – то есть возможно определение рекурсивных типов.

Как конструировать деревья? По аналогии с тем, как мы конструировали значения типа `CharOrInt`. Например:

`Leaf 1` – лист со значением 1

`BiTree 2 (Leaf 1) (Leaf 3)` – дерево из трех узлов

`BiTree 7 (Leaf 0) (BiTree 2 (Leaf 1) (Leaf 3))` – дерево из пяти узлов

Также, аналогично `CharOrInt`, производится и обработка деревьев. Например, напишем функцию, определяющую глубину дерева:

```
depth :: BiTree -> Int
depth (Leaf _) = 1
depth (BiTree _ left right) = 1 + max (depth left) (depth right)
```

Или увеличение значения всех улов дерева на единицу:

```
incValues :: BiTree -> BiTree
incValues (Leaf v) = Leaf (v+1)
incValues (BiTree v left right) =
```

```
    BiTree (v+1) (incValue left) (incValue right)
```

Или проверку равенства деревьев:

```
equalTrees :: BiTree -> BiTree -> Bool
equalTrees (Leaf x) (Leaf y) = x == y
equalTrees (BiTree x l1 r1) (BiTree y l2 r2) =
    x == y && equalTrees l1 l2 && equalTrees r1 r2
equalTrees _ _ = False
```

В этом определении стоит обратить внимание на то, что в Haskell нельзя написать образец вида `(Leaf x) (Leaf x)`, проверка равенства должна быть вынесена за границы образца.

Теперь рассмотрим ещё одну возможность при определении типов – именование полей.

Допустим, нам необходимо написать функцию, которая получает значение верхнего узла дерева. Она, конечно очень простая:

```
value :: BiTree -> Int
value (Leaf v) = v
value (BiTree v _ _) = v
```

Однако она может быть автоматически создана интерпретатором, если мы немного изменим определение типа дерева:

```
data BiTree = Leaf{ value :: Int } |
              BiTree{ value :: Int, left, right :: BiTree }
```

В этом определении автоматически создаются три функции, которые позволяют получать значения аргументов конструкторов для дерева – `value`, `left` и `right`, причем `value` применима ко всем деревьям, а `left` и `right` только к деревьям, имеющим дочерние узлы.

Обработка деревьев с помощью функций высшего порядка

Рассмотрим такие задачи, как вычисление минимального значения в дереве, вычисление суммы значений, их произведения и т.п. Они записываются следующим образом:

```
minValue :: BiTree -> Int
minValue (Leaf v) = v
minValue (BiTree v l r ) = min3 v (minValue l) (minValue r)

min3 :: Int -> Int -> Int -> Int
min3 a b c = min a (min b c)

summ :: BiTree -> Int
summ (Leaf v) = v
summ (BiTree v l r ) = summ l + v + summ r
```

```
prod :: BiTree -> Int
prod (Leaf v) = v
prod (BiTree v l r ) = prod l * v * prod r
```

Можно обратить внимание на то, что все они выглядят очень похоже и отличаются только второй строкой определения, а именно тем, как получается итоговое значение для узла дерева из значения узла и значений, вычисленных для поддеревьев. В случае минимального значения – это минимум трёх чисел, в случае суммы – сумма, в случае произведения – произведение этих значений. Раз эти функции так похожи, попробуем написать одну общую функцию, которая в зависимости от одного из аргументов могла бы вычислять любое из этих значений. Эта функция будет принимать закон, по которому комбинируются числа, и дерево, а возвращать итоговое значение для дерева:

```
reduce :: (Int -> Int -> Int -> Int) -> BiTree -> Int
```

Для листьев дерева она будет возаращать просто значения в них:

```
reduce f (Leaf x) = x
```

А для более сложных деревьев – комбинировать значения с помощью переданного закона:

```
reduce f (BiTree x l r) = f x (reduce f l) (reduce f r)
```

Эта функция принимает в качестве аргумента другую функцию (на самом деле такие функции мы уже встречали – например, `diff` и `twice`). Такие функции (оперирующие другими функциями) называются функциями высшего порядка или функционалами.

Чем же такое определение удобно? После определения такой функции наши функции `minValue`, `summ` и `prod` могут быть определены как:

```
minValue t = reduce (\x vl vr -> min3 x vl vr) t
summ t = reduce (\x vl vr -> x+vl+vr) t
prod t = reduce (\x vl vr -> x*vl*vr) t
```

Обратим внимание, что последний аргумент (`t`) передаётся неизменным в функцию `reduce`, а значит можно ещё укоротить эти определения, используя частичную применимость функций:

```
minValue = reduce (\x vl vr -> min3 x vl vr)
summ = reduce (\x vl vr -> x+vl+vr)
prod = reduce (\x vl vr -> x*vl*vr)
```

Каждая из наших трёх функций записана в одну строку – в случае, если таких функций много, это может значительно сократить код. Однако это ещё не всё. Попробуем теперь ввести над деревьями некоторое упорядочивание, которое основано на том принципе, что мы сначала сопоставляем каждому дереву некоторое число, а затем упорядочиваем деревья так же, как и соответствующие числа. Понятно, что может быть много таких упорядочиваний – например, по сумме значений, по значению в корне, по максимальному значению и т.п. Сразу напишем обобщённую функцию такого сравнения, проверяющую, является ли первое дерево больше второго, и использующую функциональный аргумент для оценки деревьев:

```
more :: (BiTree -> Int) -> BiTree -> BiTree -> Bool
more f t1 t2 = (f t1) > (f t2)
```

Сразу обратим внимание на то, что определение функции получилось очень простое. Однако используя его, мы можем получить сразу большое число функций упорядочивания по разным критериям, например:

```
moreByMin :: BiTree -> BiTree -> Bool
moreByMin t1 t2 = more minValue t1 t2
moreBySumm t1 t2 = more summ t1 t2
moreByValue t1 t2 = more value t1 t2
```

Опять же, последние два аргумента передаются в неизменном виде в функцию, возвращающую результат, а потому можно записать эти строки, используя частичное применение:

```
moreByMin = more minValue
moreBySumm = more summ
moreByValue = more value
```

Или же, если вспомнить как определены вспомогательные функции `minValue` и `summ`, то:

```
moreByMin = more (reduce (\x vl vr -> min3 x vl vr))
moreBySumm = more (reduce (\x vl vr -> x+vl+vr))
```

Таким образом, определение пары обобщённых функций высшего порядка позволяет иногда значительно уменьшить количество повторяющегося кода, однако зачастую ценой усилий, которые необходимы для его понимания.