Полиморфные функции

На предыдущих занятиях мы определяли функции, которые работали с фиксированными типами данных, однако можно было заметить, что некоторые функции могли бы быть успешно применимыми для различных типов данных. Например, рассмотрим функцию высшего порядка twice:

```
twice :: (Integer -> Integer) -> Integer -> Integer
twice f x = f (f x)
```

В принципе, она так же работала бы и для любого другого типа, поэтому нет смысла ограничивать ее типом `Integer`. Попробуем записать это функцию так, чтобы она была применима для любого типа – обозначим его `a`:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Данная запись является корректной с точки зрения языка Haskell и обозначает именно то, что нам и требовалось. Здесь `a` является переменной типа и может принимать в качестве значения любой тип. Соответственно, эту функцию можно применять для различных типов.

Применение таких функций внешне никак не отличается от применения функций с фиксированными типами аргументов. Например для `Int` ее можно применять так:

```
inc :: Int -> Int
inc x = x + 1


twice inc 1  ≡ 3
```

Для кортежей:

```
doubleVec :: (Float,Float) -> (Float,Float)
doubleVec (x,y) = (2*x,2*y)


twice doubleVec (2,5) ≡ (8,20)
```

Чтобы понять, как именно это все работает, сначала рассмотрим более простой пример функции:

```
f :: a -> a
```

Эта функция принимает значение некоторого типа и возвращает значение того же типа. Как она реализована сейчас совершенно не важно. Рассмотрим ее вызов:

```
f 1
```

Здесь функция применяется к числу (будем считать, что это число имеет тип `Int`, хотя, как станет понятно на следующем занятии не все так просто). При анализе подобного вызова производится ***сопоставление*** типа параметра функции и передаваемого значения. В процессе этого сопоставления входящие в параметр переменные типа получают некоторые значения, которые подставляются в тип результата (остаточной функции). Соответственно, в данном случае происходит сопоставление между переменной типа `a` и типом `Int`, что приводит к тому, что результат тоже становится `Int`.

Вернемся к более сложному случаю – функции `twice`. Рассмотрим применение:

```
twice inc 1
```

Поскольку параметр функции twice имеет тип `a->a`, а функция `inc` – тип `Int->Int`, то в результате сопоставления переменная типа `a` получает значение `Int`, и результат имеет тип `Int`.

Аналогично, в случае вызова `twice doubleVec (2,5)` переменная `a` получает значение `(Float,Float)`. В случае же выражения `twice doubleVec 1` набор согласований, в которых участвует переменная `a`, противоречив, и компилятор выдаст ошибку.

Учитывая частичную применимость функций мы также можем записать, что:

```
twice int :: Int -> Int
twice doubleVec :: (Float,Float) -> (Float,Float)
```

Теперь попытаемся понять, какой тип имеет выражение `twice twice`. С первого взгляда может показаться, что оно ошибочно, так как `twice` – это функция двух аргументов, а первым аргументом функции `twice` является функция одного аргумента. Однако, если вспомнить про частичную применимость, то тип функции `twice` можно записать следующим образом:

```
twice :: (a -> a) -> (a -> a)
```

Чтобы не запутаться в буквах `a` при сопоставлении будем считать, что в выражении `twice twice` первая `twice` имеет тип `(b -> b) -> (b -> b)` – это корректно, поскольку `a`, входящие в тип функции и аргумента – это ***разные*** `a`. Тогда в результате сопоставления мы получим, что `b = (a->a)`, что приведет, к тому, что тип `twice twice` будет `b -> b` или `(a->a)->(a->a)`, то есть такой же, как у исходной функции `twice`! Это может показаться необычным, но если рассмотреть семантику такой функции, то можно понять, что `twice twice` – это функция, которая применяет свой первый аргумент ко второму четыре раза. Можно показать это чуть более формально (здесь не учитывается ленивость, но это не влияет на результат вычисления):

```
twice twice f x ≡
  (twice twice f) x ≡
  (twice (twice f)) x ≡
  (twice f) ((twice f) x) ≡
  (twice f) (f (f x)) ≡
  f (f (f (f x)))
```

Теперь рассмотрим другую функцию — `applyn`, которая применяет функцию к аргументу заданное число раз. Для `Integer` мы записывали ее как:

```
applyn :: Int -> (Integer -> Integer) -> Integer -> Integer
applyn 0 _ x = x
applyn n f x = f (applyn (n-1) f x)
```

Более общий вариант может быть теперь записан как:

```
applyn :: Int -> (a -> a) -> a -> a
applyn 0 _ x = x
```

```
applyn n f x = f (applyn (n-1) f x)
```

Тут стоит обратить внимание, что:

```
applyn 2 ≡ twice :: (a -> a) -> a -> a
```

То есть результатом примененения полиморфной функции к аргументам может быть другая полиморфная функция.

Использование переменных типов позволяет записать функции, которые реализуют вычисления, применимые к различным структурам данных, один раз, вместо того, чтобы определять их для каждого типа. Если вспоминать императивные языки, то это аналогично использованию шаблонов в C++.

Типы, зависящие от параметров

Рассмотрим работу с типами данных, завяисящими от переменных типа. Начнем с кортежей. Как уже упоминалось ранее, для кортежей, состоящих из двух элементов, определены стандартные функции `fst` и `snd`. Рассмотрим их возможное определение с участием переменных типа:

```
fst :: (a,b) -> a
fst (x,_) = x


snd :: (a,b) -> b
snd (_,y) = y
```

Ранее мы определяли тип `Vector3`, который представлял трёхмерный вектор. Однако, он имел фиксированный тип координат, а иногда требуется иметь вектора целых чисел, вектора дробных чисел и т. п. Для этого можно определить тип `Vector3` следующим образом:

```
type Vector3 a = (a,a,a)
```

Здесь `a` — переменная типа, являющаяся параметром типа `Vector3`. С использованием такого определения можно определить функции, которые, например, возвращают координаты вектора:

```
vecx :: Vector3 a -> a
vecx (x,_,_) = x
vecy :: Vector3 a -> a
vecy (_,y,_) = y
vecz :: Vector3 a -> a
vecz (_,_,z) = z
```

Или функцию, которая возвращает координату по индексу:

```
vecCoord :: Int -> Vector3 a -> a
vecCoord 1 (x,_,_) = x
vecCoord 2 (_,y,_) = y
vecCoord 3 (_,_,z) = z
```

Тогда, кстати, функции `vecx`, `vecy` и `vecz` могут быть определены как:

```
vecx = vecCoord 1
vecy = vecCoord 2
vecz = vecCoord 3
```

Аналогично кортежам, можно определить алгебраические типы, зависящие от параметров. Например, определим дерево, которое может хранить значения любых типов:

```
data BiTree a = Leaf a | BiTree a (BiTree a) (BiTree a)
```

Как видно, дерево определяется так же, как и дерево, содержащее значения типа `Int`, с заменой `Int` на переменную типа `a`, которая является параметром типа.

Функции для вычисления глубины дерева и количества узлов будут выглядеть в точности так же, как и функции для дерева, содержащего числа, однако будут иметь более широкий тип:

```
depth :: BiTree a -> Int
depth (Leaf _) = 1
depth (BiTree _ l r) = 1 + max (depth l) (depth r)


size :: BiTree a -> Int
size (Leaf _) = 1
size (BiTree _ l r) = 1 + size l + size r
```

Однако, написать функцию поиска элемента в таком дереве мы пока не можем — для произвольного типа `a` не определена операция сравнения.

Классы типов

Поскольку не все типы в Haskell поддерживают все функции, необходимо иметь способ указывать в функциях, какие операции должны поддерживаться типом. Для этого Haskell предоставляет концепцию **классов типов**, которые определяют множества операций, применимых к типам. Например, типы, поддерживающие операцию сравнения на равенство относятся к стандартному классу `Eq`, поддерживающие упорядочивание — классу `Ord`, поддерживающие арифметические операции — классу `Num`.

Тут стоит обратить внимание на то, что если в императивных объектно-ориентированных языках класс — это тип, то в Haskell класс — это скорее множество типов.

Приведем некоторые операции, определенные для типов, относящихся к упомянутым трем классам:

- `Eq`: `==` (равно) и `/=` (не равно);
- `Ord`: `>`, `<`, `>=`, `<=`, `max a b` (максимум из двух значений), `min a b` (минимум);
- `Num`: `+`, `-`, `*`, `abs`, `signum`.

Теперь, имея некоторое начальное представление о базовых классах типов, рассмотрим, как их можно использовать при определении функций.

Начнем с функции возведения числа в квадрат. Понятно, что любое число можно умножить на себя и при этом результатом будет число того же типа. Соответственно для того, чтобы функция

была применима, необходимо записать тот факт, что тип её параметров и результата должен быть числом, то есть относится к классу `Num`:

```
square :: Num a => a -> a
square a = a*a
```

Как видно из примера, отношение между классом и переменной типа `a` записывается перед двойной стрелкой слева от самого типа. Эта запись называется ***контекстом*** и определяет условия, налагаемые на типы, которые могут быть значениями переменных типа, встречающихся справа.

Рассмотрим другой простой пример — функцию, которая выбирает значение максимальной из координат вектора. Понятно, что для того, чтобы можно было выбрать максимальную координату, необходимо, чтобы типы, представляющие координаты, были упорядочиваемыми, то есть относились к классу `Ord`:

```
maxCoord :: Ord a => Vector3 a  -> a
maxCoord (a,b,c) | a >= b && a >= c = a
                 | b >= a && b >= c = b
                 | otherwise = c
```

Или же, с использованием функции `max`:

```
maxCoord :: Ord a => Vector3 a  -> a
maxCoord (a,b,c) = max a (max b c)
```

Теперь можно наконец написать функцию проверки наличия заданного значения в дереве:

```
hasValue :: Eq a => BiTree a -> a -> Bool
hasValue (Leaf v) x = v == x
hasValue (BiTree v l r) x = v == x || hasValue l x || hasValue r x
```

Приведем аналогичный пример для дерева поиска:

```
data SearchTree a = EmptyTree |
   SearchTree a (SearchTree a) (SearchTree a)


hasValue :: Ord a => SearchTree a -> a -> Bool
hasValue EmptyTree _ = False
hasValue (SearchTree i t1 t2) x | i > x = hasValue t1 x
                                | i < x = hasValue t2 x
                                | otherwise = True
```

Таким образом, с использованием классов типов можно указывать требования к типам, к которым должны быть применимы те или иные функции.