Экземпляры классов типов

На прошлом занятии мы встретили понятие классов типов и научились писать функции, которые их используют. Теперь рассмотрим как связать классы типов со своими структурами данных, а также как объявить новые классы.

Начнем с класса `Eq`, который содержит операции сравнения на равенство и неравенство. Чтобы сделать эти операции применимыми для, например, бинарного дерева, необходимо объявить, что тип дерева является экземпляром класса `Eq`, и указать как именно реализуется операция сравнения.

Сначала рассмотрим это для случая неполиморфного дерева, которое объявлено следующим образом:

```
data IntBiTree = IntLeaf Int | IntBiTree Int IntBiTree IntBiTree
```

Для объявления типа экземпляром класса используется ключевое слово `instance`, которое используется следующим образом:

```
instance Eq IntBiTree where
    (==) (IntLeaf va) (IntLeaf vb) = va == vb
    (==) (IntBiTree va la ra) (IntBiTree vb lb rb) =
             va == vb && la == lb && ra == rb
    (==) _ _ = False

    (/=) a b = not (a==b)
```

Здесь первая строка сообщает интерпретатору о том, что `IntBiTree` является экземпляром класса, а дальше идет определение операций, составляющих класс: проверки на равенство и проверки на неравенство.

При определении операции равенства деревьев стоит обратить внимание на выражения `la==lb` и `ra==rb`, которые по сути являются рекурсивными вызовами определяемой операции.

Тут можно заметить, что операция `/=` должна бы выглядеть одинаково для всех типов, имеющих операцию `==` и ее можно было бы определить в самом классе `Eq`. Так и сделано, и потому ее определение здесь не обязательно. Кроме того, имеет смысл обратить внимание на возможность записи операций в инфиксной форме (подобно тому, как они могут быть вызваны).

С учетом этих замечаний можно записать определение экземпляра класса следующим образом:

```
instance Eq IntBiTree where
    (IntLeaf va) == (IntLeaf vb) = va == vb
    (IntBiTree va la ra) == (IntBiTree vb lb rb) =
             va == vb && la == lb && ra == rb
    _ == _ = False
```

Теперь перейдем к определению экземпляра класса для полиморфного дерева, определенного следующим образом:

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

Оно полностью аналогично приведенному выше и отличается только первой строкой:

```
instance Eq a => Eq (BiTree a) where
    (Leaf va) == (Leaf vb) = va == vb
    (BiTree va la ra) == (BiTree vb lb rb) =
             va == vb && la == lb && ra == rb
    _ == _ = False
```

В первой строке задано уже более сложное условие, которое можно прочитать как “для каждого типа `a`, который является экземпляром класса `Eq` тип `BiTree a` тоже является экземпляром класса `Eq`”. Или – деревья, содержащие сравнимые элементы можно сравнивать. То есть так же, как и для функций, можно определять ограничения на типы, которые могут быть объявлены экземплярами классов.

Несколько слов стоит сказать о роли классов в языке. Помимо того, что они позволяют объединять схожие типы в некоторые множества, они предоставляют второй вид полиморфизма в Haskell, аналогичный полиморфизму в объектно-ориентированных языках. Действительно, в отличие от параметрического полиморфизма, где одна и та же функция работает одинаково для всех типов, в случае классов мы имеем механизм, позволяющий одной и той же функции **по-разному** работать для различных типов. Это аналогично использованию виртуальных методов в объектно-ориентированных языках программирования.

Класс Enum

Перед рассмотрением числовых классов рассмотрим еще один класс. Мы раньше определяли перечислимые типы, и уже отмечалось, что Haskell по умолчанию не предоставляет методов преобразования перечисляемых значений в числа и обратно. Однако такую возможность было бы неплохо иметь. Для этого используется класс `Enum`, в котором определяются следующие основные операции:

```
toEnum :: Int -> a
fromEnum :: a -> Int
succ, prev :: a -> a
```

Операции `toEnum` и `fromEnum` осуществляют преобразование из числа в перечисляемый тип и обратно, `succ` и `prev` осуществляют получение следующего и предыдущего элемента.

Реализуем этот класс для типа `Day`, который будет представлять день недели:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Определение экземпляра класса выглядит так:

```
instance Enum Weekday where
    fromEnum Mon = 0
    fromEnum Tue = 1
    fromEnum Wed = 2
    fromEnum Thu = 3
```

```
    fromEnum Fri = 4
    fromEnum Sat = 5
    fromEnum Sun = 6

    toEnum 0 = Mon
    toEnum 1 = Tue
    toEnum 2 = Wed
    toEnum 3 = Thu
    toEnum 4 = Fri
    toEnum 5 = Sat
    toEnum 6 = Sun

    succ a = toEnum ((fromEnum a + 1) `mod` 7)
    pred a = toEnum ((fromEnum a + 6) `mod` 7)
```

Здесь функции `fromEnum` и `toEnum` определены простым перечислением возможных значений, а `succ` и `pred` – путем перевода в число и обратно (хотя можно было бы конечно обойтись перечислением значений).

Определение классов

Теперь можно перейти к тому, как определяются классы. В качестве примера рассмотрим объявление стандартного класса `Eq`, о котором уже много было сказано выше:

```
class  Eq a  where
    (==), (/=)  ::  a -> a -> Bool

    x /= y  = not (x == y)
    x == y  = not (x /= y)
```

Как видно, для определения класса используется ключевое слово `class`. Такое объявление говорит о том, что тип `a` может являться экземпляром класса `Eq` если у него имеются операции `==` и `/=` заданного типа. После первой строки идет определение типов операций, которые имеются в классе, а в конце – реализация некоторых операций по умолчанию. В данном случае операции `==` и `/=` определены друг через друга, и поэтому при определении экземпляра класса достаточно определить только одну из них.

Если вернуться к классу Enum, то он определяется следующим образом:

```
class Enum a where
    succ, pred     :: a -> a
    toEnum         :: Int -> a
    fromEnum       :: a -> Int
```

Теперь рассмотрим описание какого-то нового класса. Например, вспомним, что мы описывали полиморфные типы бинарного дерева и дерева поиска. Для каждого из этих типов мы определяли функцию проверки наличия элемента в дереве. Можно выделить класс `Set`, для которого будет определена проверка вхождения элемента в множество, и за счет этого иметь возможность использовать различные типы деревьев единообразно:

```
class Set m where
    contains :: Eq a => m a -> a -> Bool
```

Тогда, если у нас есть полиморфное бинарное дерево:

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

То можно объявить его экземпляром класса `Set`, реализовав функцию `contains`:

```
instance Set BiTree where
    contains (Leaf v) x = v == x
    contains (BiTree v l r) x = v == x ||
                                    contains l x || contains r x
```

Здесь стоит обратить внимание на то, что при объявлении класса и экземпляра для типов, предполагающих некоторые параметры, можно использовать их непараметризованную форму.

Класс Num

Теперь перейдем к рассмотрению стандартного класса `Num`. Он определен в стандартной библиотеке следующим образом:

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
```

Как видно, в нем определены основные операции, которые применимы ко всем числам, а также функции для получения отрицания числа, его модуля и знака. Отдельное место занимает функция `fromInteger`, которая преобразует `Integer` в число описываемого типа.

Слева от `=>` написаны ограничения на тип `a`, которые должны быть выполнены для того, чтобы класс мог быть применим к некоторому типу. В данном случае, эти ограничения состоят в том, что любой тип, принадлежащий классу `Num` должен также принадлежать классам `Eq` и `Show`. В таких случаях говорят, что класс `Num` является наследником классов `Eq` и `Show`.

Теперь попробуем определить тип, представляющий натуральное число. Для того, чтобы определить сам тип, воспользуемся конструкцией `data`, определив один конструктор:

```
data Natural = Natural Integer
```

Теперь необходимо реализовать класс `Num`, но он требует наличия класса `Eq` (`Show` пока проигнорируем, до него дело еще дойдет), поэтому реализуем сначала `Eq`:

```
instance Eq Natural where
```

```
    (Natural a) == (Natural b) = a == b
```

Теперь можно определить и экземпляр класса `Num`:

```
instance Num Natural where
    (Natural a) + (Natural b) = fromInteger (a + b)
    (Natural a) - (Natural b) = fromInteger (a - b)
    (Natural a) * (Natural b) = fromInteger (a * b)

    fromInteger x | x > 0 = Natural x
                  | otherwise = undefined

    abs a = a
    signum a = 1
    negate a = undefined
```

Здесь функция `fromInteger` проверяет, что число, которое мы хотим считать натуральным, действительно больше нуля, а если нет, то возвращает `undefined`. Операции реализованы по схеме “распаковать число – выполнить операцию – преобразовать обратно в натуральное”.

Теперь рассмотрим, как можно эти натуральные числа использовать. Для примера напишем функцию инкремента натурального числа. Первый вариант:

```
incNat :: Natural -> Natural
incNat (Natural a) = Natural (a + 1)
```

Здесь мы распаковываем число, а потом запаковываем его заново. Но поскольку, для натурального числа определена операция сложения можно записать ее так:

```
incNat :: Natural -> Natural
incNat a = a + Natural 1
```

Здесь мы прибавляем к натуральному числу натуральную единицу. Но это не все. Можно не использовать конструктор явно, а вызвать функцию `fromInteger`:

```
incNat :: Natural -> Natural
incNat a = a + fromInteger 1
```

Но, для такой записи есть более короткая форма:

```
incNat :: Natural -> Natural
incNat a = a + 1
```

Почему это работает? Любой целочисленный литерал, который встречается в программе на самом деле воспринимается интерпретатором как вызов функции `fromInteger` с соответствующим аргументом. Таким образом, ***последние две записи эквивалентны***. Именно за счет этого можно передавать целые числа в качестве аргументов в функции, принимающие `Float` или другие числовые типы. И именно это объясняет то, почему полиморфный факториал с предыдущего занятия работал:

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

В данном случае, `1` и `0` автоматически преобразуются к нужному типу за счет использования функции `fromInteger`.

Конструкция newtype

Теперь немного прервемся для того, чтобы рассмотреть другую конструкцию, используемую для объявления подобных типов в Haskell. Она называется `newtype` и используется аналогично `data` с одним конструктором:

```
newtype Natural = Natural Integer
```

Работа с типами, определенными через нее, производится абсолютно также, за исключением одного отличия, связанного с обработкой неопределенных значений: Если `Natural` объявлен с использованием `data`, то `Natural undefined` и `undefined` это не одно и то же, в случае же `newtype` – `Natural undefined` это то же самое, что просто `undefined`.

Это может проявится в следующем коде:

```
case undefined of
  Natural _ = 0
```

В данном случае, если `Natural` объявлено через `data`, то результат будет `undefined`, а если через `newtype`, то результатом будет `0`.

Класс Ord

Рассмотрим другой стандартный класс – `Ord`, который описывает упорядочиваемые типы:

```
data Ordering = EQ | LT | GT
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y | x == y    = EQ
                | x < y     = LT
                | otherwise = GT

    x <= y  = compare x y /= GT
    x <  y  = compare x y == LT
    x >= y  = compare x y /= LT
```

```
    x >  y  = compare x y == GT

    max x y | x <= y    =  y
            | otherwise =  x
    min x y | x <= y    =  x
            | otherwise =  y
```

В нем используется перечислимый тип `Ordering`, который задает отношение между двумя значениями (равно, меньше, больше). В классе `Ord` для всех функций имеется реализация по умолчанию, однако, если посмотреть внимательно, то можно понять, что при реализации класса достаточно реализовать только `compare` или оператор `<=` (поскольку оператор `==` уже реализован, так как класс наследуется от `Eq`).

Теперь определим экземпляр класса Ord для типа Natural:

```
instance Ord Natural where
    (Natural a) <= (Natural b) = a <= b
```

Используя это определение, можно написать, например, функцию вычисления наибольшего общего делителя:

```
nod :: Natural -> Natural -> Natural
nod a b | a == b = a
        | a > b = a - b
        | b > a = b - a
```

Автоматическая генерация экземпляров классов

Можно обратить внимание на то, что большинство реализаций классов, которые приводились здесь, достаточно тривиальны. Большинство из них могут быть автоматически сгенерированы компилятором для простых типов.

Рассмотрим, как можно автоматически сгенерировать операции сравнения на равенство, на примере нашего бинарного дерева:

```
data IntBiTree =
  Leaf Int | IntBiTree Int IntBiTree IntBiTree deriving Eq
```

Если необходимо добавить еще упорядоченность, то:

```
data IntBiTree =
  Leaf Int | IntBiTree Int IntBiTree IntBiTree deriving (Eq,Ord)
```

Аналогично можно автоматически генерировать экземпляры и для полиморфных типов:

```
data BiTree a =
  Leaf a | BiTree a (BiTree a) (BiTree a) deriving (Eq,Ord)
```

В данном случае, понятно, что для того, чтобы можно было сравнивать или упорядочивать деревья, элементы дерева должны быть сравнимыми и упорядоченными. В случае

вышеприведенного объявления, этот факт проверяется в момент, когда тип дерева возникает в программе. Однако, более правильным будет описание типа следующим образом:

```
data (Eq a, Ord a) => BiTree a =
  Leaf a | BiTree a (BiTree a) (BiTree a) deriving (Eq,Ord)
```

В этом случае ограничения заданы при объявлении типа.

Экземпляры могут быть автоматически сгенерированы для классов `Eq`, `Ord`, `Enum`, `Bounded`, `Show` и `Read`, более подробную информацию можно найти в соответствующем разеделе спецификации Haskell (http://www.haskell.ru/derived.html).

Классы Show и Read

Теперь рассмотрим пару классов, полезных при работе в интерактивной консоли. Если попытаться в интерактивной консоли вычислить простейшее выражение, включающее определенные нами ранее типы, то можно получить следующую ошибку:

```
Main> (Leaf 1)
ERROR - Cannot find "show" function for:
*** Expression : Leaf 1
*** Of type    : BiTree Integer
```

В тексте ошибки сообщается, что интепретатор не может найти функцию `show`, которая необходима для того, чтобы преобразовать значение в строку.

Для того, чтобы решить эту проблему и иметь возможность выводить на печать значения сложного типа, необходимо объявить для этого типа экземпляр класса `Show`, содержащего функцию `show` и объявленный в стандартной библиотеке (здесь упрощенное объявление):

```
class Show a where
    show      :: a -> String
```

Поскольку работу со строками мы не рассматривали, то вместо непосредственного объявления экземпляра класса мы будем использовать его автоматическую генерацию с помощью ключевого слово `deriving`:

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

Дерево, объявленной таким образом, уже может быть выведено на печать:

```
Main> (Leaf 1)
Leaf 1
```

Для полноты картины стоит упомянуть здесь и класс, позволяющий выполнить обратную задачу – построить значение типа на основе его строкового представления. Этот класс также находится в стандартной библиотеке и называется `Read`, его упрощенное объявление выглядит как:

```
class Read a where
    read      :: String -> a
```

Его экземпляр также может быть автоматически сгенерирован:

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

Отображение значений. Класс Functor

На предыдущих занятиях мы рассматривали функцию `mapTree`, которая осуществляла преобразование дерева путем применения заданной в качестве аргумента функции к каждому узлу дерева. Например:

```
Main> mapTree (\x -> x+1) (BiTree 3 (Leaf 1) (Leaf 5))
BiTree 4 (Leaf 2) (Leaf 6)
```

Стоит напомнить определение этой функции:

```
mapTree :: (a -> a) -> BiTree a -> BiTree a
mapTree f (Leaf v) = Leaf (f v)
mapTree f (BiTree v l r) = BiTree (f v) (mapTree f l) (mapTree f r)
```

Подобные функции очень часто применяются в функциональном программировании для различных структур данных и выполняют задачу преобразования сложного значения путем применения функции к его составным частям с сохранением структуры сложного значения. Чтобы понять, что в данном случае понимается под “сохранением структуры”, рассмотрим два примера.

Сначала рассмотрим функцию `id`, которая определена следующим образом:

```
id :: a -> a
id x = x
```

Как видно, эта функция просто возвращает свой аргумент. Логично предположить, что если мы передадим эту функцию в качестве превого аргумента в `mapTree`, то в результате мы должны получить исходное дерево:

```
Main> mapTree id (BiTree 3 (Leaf 1) (Leaf 5))
BiTree 3 (Leaf 1) (Leaf 5)
```

Логично считать, что такое свойство должно выполняться и для всех остальных сходных преобразований.

Теперь рассмотрим такой пример – мы можем удвоить значения в узлах дерева, а затем прибавать к каждому 1. Можно предложить два основных способа это сделать:

```
mapTree (+1) (mapTree (*2) t)
--Сначала удваиваем все значения, потом увеличиваем на 1
mapTree (\x -> x*2+1) t
-- Сразу удваиваем все значения и увеличиваем на 1
```

Логично потребовать чтобы оба способа всегда приводили к одному и тому же результату.

Таким образом, мы получили два основных требования для функций, которые осуществляют такие преобразования:

```
mapTree id t = t
mapTree f (mapTree g t) = mapTree (\x -> f (g x)) t
```

Эти требования и будем считать тем, что обеспечивает сохранение структуры. Теперь попробуем несколько обобщить нашу функцию преобразования деревьев. Сейчас она принимает в качестве аргумента функцию, тип аргумента и результата которой совпадают, но это ограничение можно отбросить. Действительно, ничто не мешает так же преобразовывать дерево, в узлах которого расположены `Float` в дерево целых чисел путем применения операции округления к каждому узлу:

```
Main> mapTree round (BiTree 32.6 (Leaf 1.4) (Leaf 5.1))
BiTree 33 (Leaf 1) (Leaf 5)
```

Чтобы такое применение стало допустимым, необходимо немного изменить тип функции. Теперь она будет выглядеть так:

```
mapTree :: (a -> b) -> BiTree a -> BiTree b
mapTree f (Leaf v) = Leaf (f v)
mapTree f (BiTree v l r) = BiTree (f v) (mapTree f l) (mapTree f r)
```

Стоит отметить, что оба требования, обеспечивающие сохранение структуры, верны и для этой функции.

Поскольку такие функции являются очень распространенными в Haskell, предусмотрен стандартный класс `Functor`, в котором такая функция определена:

```
class  Functor f  where
    fmap :: (a -> b) -> f a -> f b
```

При этом рекомендуется (но не требуется), чтобы при определении экземпляров этого класса функция `fmap` удовлетворяла выщеописанным условиям сохранения структуры. Поскольку наша функция `mapTree` как раз такой и является, мы можем объявить экземпляр класса `Functor` для типа `BiTree`:

```
instance Functor BiTree where
  fmap = mapTree
```

Теперь наше дерево можно использовать в обобщенных функциях, которые применимы ко всем подобным структурам данных.