ТФП, язык Haskell

Лекция 8

Контейнерный тип данных

Контейнерный тип данных – такой тип, значения которого содержат в себе объекты других типов.

Пример:
Just 10*
[1,2,3]

(каковы типы у этих объектов?)

Класс Functor

Определение класса Functor:

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

Исходя из типа функции fmap можно сделать вывод о том, что f – это некоторый контейнерный тип.

Класс Functor

Тип функции fmap:

```
fmap :: ( a -> b ) -> f a -> f b
```

«`fmap` – функция, принимающая на вход два аргумента, первый из которых – функция, определённая на значениях типа `a` и принимающая значения на типе `b`, второй аргумент – некий контейнер типа `f`, содержащий значения типа a и результат – контейнер типа `f`, содержащий значения типа `b`»

Класс Functor, интуиция (1/2)

```
fmap :: ( a -> b ) -> f a -> f b
```

Типом `f` является контейнерный тип (например, список, дерево, `Maybe`) содержащий значения типа `a`.

Мы хотим применить к элементам внутри этого контейнера функцию (первый аргумент `fmap`) и в результате получить новый контейнер, содержащий результат данного преобразования.

Класс Functor, интуиция (1/2)

```
fmap :: ( a -> b ) -> f a -> f b
```

В качестве примера можно вспомнить функцию map, которая применяет функцию (свой первый аргумент) ко всем элементам в списке (второй аргумент) и возвращает список, содержащий результаты.

```
map ( (+) 1 ) [1,2,3]
```

Класс Functor, интуиция (2/2)

```
fmap :: ( a -> b ) -> ( f a -> f b )
```

Еще один интуитивный подход к функции fmap. В Haskell все функции каррированы. В данном случае можно думать о функции fmap, как о функции, которая принимает на вход «обычную» (определённую над некими типами) функцию и в качестве результата производит её вариант для работы с данным контейнерным типом.

Класс Functor, законы

```
fmap :: ( a -> b ) -> ( f a -> f b )
```

Реализация функции fmap должна удовлетворять двум законам:

```
fmap id = id*
fmap ( g . h ) = fmap g . fmap h
```

Таким образом, функтор должен сохранять структуру композиции морфизмов.

*каков тип у функции id? Что она делает?

Класс Functor, go ahead

```
fmap :: ( a -> b ) -> ( f a -> f b )

{--
   Проверьте, удовлетворяет ли такое определение
   функции fmap законам, приведённым выше.
-}

instance Functor [] where
   fmap _ [] = []
   fmap g (h:t) =  ( g h ) : ( g h ) : ( fmap g t )
```

Класс Functor, go ahead 2

```
fmap :: ( a -> b ) -> ( f a -> f b )

{--
   Проверьте, удовлетворяет ли такое определение
   функции fmap законам, приведённым выше.
--}
instance Functor Maybe where
   fmap g ( Just a ) = Just ( g a )
   fmap _ ( Nothing ) = Nothing

{--
   Возможно ли реализовать функцию fmap для Maybe так,
   чтобы она не удовлетворяла законам? Если да, то
   приведите конструктивный пример.
--}
```

A few ideas

Предположим, что у вас есть две функции типа

```
f,g :: Int - > Int
f = (+) 1
g = (+) 2
```

Вы бы хотели следить за вычислениями, где используются эти функции, например, используя некую отладочную информацию.

A few ideas

```
f,g :: Int - > Int
f = (+) 1
g = (+) 2
```

К примеру, можно определить новые функции, которые вместе с результатом исполнения f и будут возвращать строку с информацией о процессе вычисления.

```
f',g' :: Int -> (Int, String)
f' x = ( f x, «f was called»)
g' x = ( g x, «g was called»)
```

A few ideas

Для f и g очень легко определяется композиция (f.g), но что делать с композицией f' и g' (тип результатов этих функций не совпадает с типом параметров)?

Скорее всего, вам бы пришло в голову написать такой код:

```
test x = let ( y, s) = g' x
         let ( z, s' ) = f' y in ( z, s ++ s' )
```

A few ideas

Скорее всего, вам бы пришло в голову написать такой код:

```
test x = let ( y, s) = g' x
             let ( z, s' ) = f' y in ( z, s ++ s' )
```

Каждый раз, когда вам понадобилось бы использовать композицию f' и g'.

Что утомительно.

A few ideas

```
test x = let ( y, s) = g' x
           let ( z, s' ) = f' y in ( z, s ++ s' )
```

Почему бы не определить функцию, которая бы делала это за нас?

```
bind::(Int->(Int,String))->(Int,String)->(Int,String)
```

A few ideas

```
bind::(Int->(Int,String))->(Int,String)->(Int,String)
```

Что бы мы хотели от этой функции? Она, очевидно, должна связывать две функции. (См. Предыдущий слайд).

Реализуйте функцию bind.

```
Hint:
>let h' = bind f' . g'
>h' 10
(13,«f was calledg was called»)
```

A few ideas

Но что делать, если в вычислениях, где участвуют функции f' и g' мы хотим использовать значения типа Int?

Нам понадобится функция, которая переводит значение типа Int в значение типа (Int, String).

```
unit :: Int - > ( Int, String )
```

Реализуйте функцию unit. (hint: строка может быть константной)

A few ideas

Несколько примеров:

```
Prelude> unit 10
(10,"unit")
Prelude> (bind f' . ( unit ) ) 10
(11,"f was calledunit")
Prelude> bind f' (unit 10 )
(11,"f was calledunit")
```

A few ideas

Вообще, можно определить контейнерный тип

```
type Dbg a = ( a, String )
```

Тогда тип функции bind:

```
bind:: ( a -> Dbg a ) -> Dbg a -> Dbg a
```

Тип unit:

```
unit :: a -> Dbg a
```

A few ideas

```
bind :: ( a -> Dbg a ) -> Dbg a -> Dbg a
unit :: a -> Dbg a
```

А теперь давайте посмотрим на класс Monad:

```
class Monad m where
   (>>=) :: m a -> ( a -> m b ) -> m b
   (>>) :: m a -> m b -> m b
   return :: a -> m a
```

A few ideas

Мы изобрели монаду. Очевидно, что тип нашей функции bind с точностью до порядка аргументов совпадает с типом (>>=), а тип unit с совпадает с типом return.

Можно говорить о том, что мы реализовали

Instance Monad Dbg.

A few further ideas

Если вы попытаетесь описать вычислительные процессы, в которых участвуют функции типа

```
Complex Float -> [ Complex Float ]
```

Которые работают с типом комплексных чисел, вы с большой вероятностью изобретёте монады ещё раз.

A few further ideas

Если вы попытаетесь описать вычислительные процессы, в которых участвуют функции, работающие с генератором случайных чисел, вы с большой вероятностью изобретёте монады ещё раз.

Monads. Now what?

На данный момент мы поняли, что монады позволяют связывать функции в вычислительные процессы.

Теперь нам предстоит ответить на вопросы:

1) зачем и что из этого можно получить?
2) каким законам должны подчиняться наши реализации монад?

Это все будет на следующей лекции.