

Если вы видите что-то необычное, просто сообщите мне.

Аппликативные функторы

Добро пожаловать во вторую часть серии о монадах и других функциональных структур. Мы продолжим готовить собирать нашу базу изучая идеи аппликативных функторов. Если вы всё еще не имеете твердое понимание функторов, пересмотрите первую часть этой серии. Если вы считаете, что уже готовы к монадам, то можете смело переходить к части 3.

В этой части приведенные примеры можно будет опробовать на GHCi.

Функторы становятся короткими

В первой части, мы обсудили функтор типа класса. Мы нашли, то что это позволяет нам запустить преобразования данных в зависимости от того, во что обернуты данные. Не важно, являются ли наши данные `List`, `Maybe`, `Either` или даже свой собственный тип, мы можем просто вызывать `fmap`. Однако, что случится когда мы попробуем объединить обернутые данные? Для примера, если мы попробуем произвести эти вычисления с помощью GHCi, мы получим ошибку типа:

```
>> (Just 4) * (Just 5)
>> Nothing * (Just 2)
```

Могут ли функции помочь нам тут? Мы можем использовать `fmap` чтобы обернуть умножение с помощью частичной обертывания `Maybe` значения:

```
>> let f = (*) <$> (Just 4)
>> :t f
f :: Num a => Maybe (a -> a)
>> (*) <$> Nothing
```

Nothing

Это дает частичную функцию обернутую в `Maybe`. Но мы до сих пор не можем развернуть это и применить к `Just 5` в общем стиле. Поэтому нам нужно обратиться к коду специально для типа `Maybe`:

```
funcMaybe :: Maybe (a -> b) -> Maybe a -> Maybe b
funcMaybe Nothing _ = Nothing
funcMaybe (Just f) val = f <$> val
```

Это очевидно не будет работать с другими типами функторов.

Приложения в помощь

То что такое аппликативные типы классов, говорят две главные функции:

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

Чистая функция принимает какое-то значение и обортывает его в минимальный контекст. Функция `<*>` вызывает последующее приложение, которое принимает 2 параметра. Первый - принимает функцию обернутую в контекст. Второе - обернутое значение. Вывод - результат применения функции к значению, преобразованное в контексте. Экземпляр называется аппликативный функтор так как он позволяет нам применять обернутую функцию. Так как последующее применение принимает обернутую функцию, мы часто начинаем с обертки чего-то чистого или `fmap`. Это будет понятнее на примерах.

Для начала представим перемножение `Maybe` значений. Если мы умножаем на постоянное значение, мы можем использовать функторный подход. Но мы можем так же использовать аппликативный подход обернув постоянную функцию в чистую и затем использовать последовательное применение:

```
>> (4 *) <$> (Just 5)
Just 20
>> (4 *) <$> Nothing
Nothing
```

```
>> pure (4 *) <*> (Just 5)
Just 20
>> pure (4 *) <*> Nothing
Nothing
```

Теперь если мы хотим умножить 2 `Maybe` значения, мы начинаем оборачивать простую функцию произведения в чистую. Затем последовательно применяем оба `Maybe` значения:

```
>> pure (*) <*> (Just 4) <*> (Just 5)
Just 20
>> pure (*) <*> Nothing <*> (Just 5)
Nothing
>> pure (*) <*> (Just 4) <*> Nothing
Nothing
```

Реализация аппликативов

По этим примерам, мы можем сказать, что экземпляры Аппликативов для `Maybe` реализованны точно, как мы ожидаем. Чистая функция просто оборачивает значение с помощью `Just`. Затем связывает вещи вместе, если другие функции или значения будут `Nothing`, мы просто выводим `Nothing`. В противном случае применяем функцию к значению и переоборачиваем с помощью `Just`.

```
instance Applicative Maybe where
  pure = Just
  (<*>) Nothing _ = Nothing
  (<*>) _ Nothing = Nothing
  (<*>) (Just f) (Just x) = Just (f x)
```

Экземпляр аппликатива для `List` будет немного интереснее. Он может вести себя на так как мы ожидаем.

```
instance Applicative [] where
  pure a = [a]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

Чистая функция - то что мы ожидаем. Мы принимаем значение и оборачиваем его как одиночку в список. Когда мы связываем операции, мы принимаем `LIST` функций. Мы должны ожидать применения каждой функции к значению в соответствующей позиции. Однако, на самом деле мы применяем функцию из первого списка к каждому значению из второго. Когда у нас только одна функция, этот результат имеет понятное поведение. Но когда у нас несколько функций, появляется отличие.

```
>> pure (4 *) <*> [1,2,3]
[4,8,12]
>> [(1+), (5*), (10*)] <*> [1,2,3]
[2,3,4,5,10,15,10,20,30]
```

Тут легко сделать определенные операции, как нахождение попарных результатов двух списков:

```
>> pure (*) <*> [1,2,3] <*> [10,20,30]
[10,20,30,20,40,60,30,60,90]
```

Вы возможно гадаете, как мы сделаем параллельное применение функторов. Например, мы можем хотеть использовать второй список из примера выше, но иметь результат `[2,10,30]`. Для этого есть конструкт под названием `ZipList`, это новый тип вокруг списка, для которого поведение экземпляра аппликатива и предусмотрено.

```
>> import Control.Applicative
>> ZipList [(1+), (5*), (10*)] <*> [5,10,15]
ZipList {getZipList = [6,50,150]}
```

Выводы

Если все это кажется непонятным, не бойтесь вернуться к части 1 и убедиться, что у вас есть четкое понимание того, что такое функторы. Если вам всё ясно, вы готовы перейти к части 3, где мы наконец испачкаемся монадами.

Все эти идея гораздо проще понять если попытаться исполнить код из примеров самостоятельно.

Revision #2

Created 11 March 2022 05:34:43 by gasick

Updated 5 September 2022 19:19:49 by gasick