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

Законы Монад

Добро пожаловать в заключительную часть серии о монадах в Haskell. Сейчас мы уже знаем большишнство идей лежащих в основе зная их тонкости для использования в программах. Но есть еще абстрактные идеи, которые нам нужно изучить, которые связанны со всеми этими структурами. Это записи структурных "законов". Эти правила для typeclass должны выполняться чтобы пересекаться с ожиданиями других программистов.

Жизнь без законов

Помните, что Haskell отражает каждый абстрактный класс с помощью type class. Каждый из этих type class имеет одину или две главные функции. Поэтому, каждый раз реализуя эти функции и её проверки типов, мы получаем функтор/аппликатив/монаду, правильно?

Не совсем. Да, ваша программа будет собираться и у вас будет возможность использовать её объекты. Но это не значит, что ваш объект следует математическим конструктам. Если нет, ваш объект не будет полноценным для других программистов. Каждый type class имеет свои законы. Для примера, давайте вернемся к `GovDirectory` типу, который мы создавали в статье про функторы. Предположим мы сделали различные объекты функторов:

```
data GovDirectory a = GovDirectory {
  mayor :: a,
  interimMayor :: Maybe a,
  cabinet :: Map String a,
  councilMembers :: [a]
}

instance Functor GovDirectory where
  fmap f oldDirectory = GovDirectory {
    mayor = f (mayor oldDirectory),
    interimMayor = Nothing,
```

```
    cabinet = f <$> cabinet oldDirectory,
    councilMembers = f <$> councilMembers oldDirectory
  }
```

Насколько видно, это нарушает один из законов функтора. В этом случае, это будет не настоящий функтор. Его поведение должно смущать любого программиста пытающегося его использовать. Мы должны позаботиться о том, чтобы убедиться что наш экземпляр имеет смысл. Как только, вы это почувствуете для `type class`, значит вы сделали экземпляр по правилу. Не переживайте если вас что-то смущает. Эта статья очень математичка, и вы не сразу поймете, все идеи, что тут предложены. Вы можете понять и использовать эти классы без знания этих законов. Ну что же, окунемся без суеты в эти законы.

Законы функторов

Есть два закона функтороов. Первый - закон идентичности. Мы посмотрим на некотороый вариант этой идеи для каждого из этих `type class`. Вспомните, как `fmap` функция работаетс содержанием. Если мы применим нашу функцию идентичности к контейнеру, результатом будет тот же объект.

```
fmap id = id
```

Другими словами, наш функтор не должен применять какие-то дополнительные преобразования или сторонние эффекты. Он должен только применять функцию. Второй закон это композиционный. Он гласит, что реализация нашго функтора не должна ломать идею нашей функции.

```
fmap (g . f) = fmap g . fmap f

-- For reference, remember the type of the composition operator:
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

С другой стороы, мы можем собрать две функции, и объединить результат в функции поверх контейнера. С другой стороны, мы применяем первую функцию, получаем результат, и применяем вторую функции поверх. Второй закон говорит, что результаты должны быть одинаковыми. Звучит это сложно. Но вам не нужно переживать. Скорей всего елсли вы

сломаете закон композиции в Haskell, скорей всего вы сломаете и закон идентичности.

У нас всего два закона, поэтому двинем дальше.

Аппликативные законы

Аппликативные функторы - это немного сложнее чем кажется. Они имеют 4 различных закона. Первый достаточно просто. Это еще один закон идентичности:

```
pure id <*> v = v
```

Слева - обертка для идентичной функции. Затем мы применяем её к контейнеру. Закон аппликативной идентичности говорит, что в результате должен быть тот же объект. Достаточно просто.

Второй закон это закон гомоморфизма. Представим, мы оборачиваем функцию и другие объекты в чистые. Мы можем затем применить обернутую функцию поверх нормального объекта, и затем оберунть их в чистые. Закон гомоморфизма говорит, что эти результаты должны быть одинаковы.

```
pure f <*> pure x = pure (f x)
```

Мы должны увидеть чистый шаблон. Поверх этого можно сказать, что большая часть этих законов гласит, что `type class` это контенеры. Фнукция `type class` не должна иметь сторонних эффектов. Все они, что они должны - облегчать обертывание, развертывание и проеобразование данных.

Третий закон - закон обмена. Он по-сложнее. Закон говорит, что от порядка оборачивания ничего не должно зависеть. С одной стороны, мы применяем любой аппликтор над обернутым в чистую функцию объектом. С другой - первое мы применяем функцию к объекту как к аргументу. Затем применяем её к первому аппликативу. Должно получиться одно и то же.

```
u <*> pure y = pure ($ y) <*> u
```

Последний закон аппликативности, копирует второй закон функтора. Это закон композиции. Он гласит, что композиция функторов не должна влиять на результат.

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

Явное число законов, может быть переполняющим. Однако, экземпляр который вы создадиде скорей всего будет следовать законам. Двигаемся дальше!

Законы монад

У монад есть три закона. Первые два это просто законы идентичности. Как и в прошлые разы.

```
return a >>= f = f
m >>= return = m
```

Есть левая и правая части. Они утверждают, что единственное что можно делать функции это оборачивать объект(знакомо?). Нельзя изменять данные как угодно. Главный вывод такой: что ниже приведнный пример кодов одинаков.

```
func1 :: IO String
func1 = do
  str <- getLine
  return str

func2 :: IO String
func2 = getLine
```

Третий закон звучит интереснее. Он говорит нам, что асоциативность хранится внутри монад.

```
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

Но мы видим этот третий закон имеет паралельные структуры с другими композиционными законами. В первом случае, мы применяем две функции в два захода. Во втором случае, мы собирае сначала функцию, и только уже потом применяем результать. Они должны быть

одинаковы.

В результате, есть две идеи из всех законов. Первый, идентичность должна сохраняться и в обернутых функциях, как чистых так и в возвращяемых. Второе, функция композиции должна храниться во всех структурах.

Проверка законов.

Как я говорил, большая часть экземпляров, которые вы прошли, будут естественно следовать правилам. С опытом использования различных типо классов, это будет становится правдой. Haskell отличный инструмент проверки ваших экземпляров проходищих определенный закон.

Эта утилита `QuickCheck`. Она может принимать любое правило, создавать множество разных случаев тестирования в нашем экземпляре функтора `GovDirectory`. Посмотрим, как `QuickCheck` доказывает свое начальное падение, и полный успех. Для начала нужно реализовать типо класс над нашим типом. Мы можем сделать это вместе с внутренним типом `Arbtrary`, такой как встроенный тип `string`. Затем мы будем использовать все другие экземпляры `Arbitrary`, которые существуют вокруг нашего типо класса.

```
instance Arbitrary a => Arbitrary (GovDirectory a) where
  arbitrary = do
    m <- arbitrary
    im <- arbitrary
    cab <- arbitrary
    cm <- arbitrary
    return $ GovDirectory
      { mayor = m
      , interimMayor = im
      , cabinet = cab
      , councilMembers = cm
      }
```

Как только вы это выполните, вы можете описать тестовый случай для частного правила. Тогда, мы проверяем идентичность функции для функтора.

```
main :: IO ()
main = quickCheck govDirectoryFunctorCheck

govDirectoryFunctorCheck :: GovDirectory String -> Bool
govDirectoryFunctorCheck gd = fmap id gd == gd
```

Теперь, давайте проверим на сломанном экземплеря, приведенном выше. Мы можем увидеть, что простой тест упадет.

```
*** Failed! Falsifiable (after 2 tests):
GovDirectory {mayor = "", interimMayor = Just "\156", cabinet = fromList [("","")],
councilMembers = []}
```

Сообщение уточняет нам что тест `arbitrary` экземпляра не пройден. Теперь предположим правильнй экземпляр:

```
interimMayor = f <$> (interimMayor oldDirectory),
```

Тест пройден!

```
+++ OK, passed 100 tests.
```

Выводы

Так мы можем обертывать наши монады! Помните, что если любая из этих идей до сих пор вас сммущает, не переживайте, и перечитывайте информаци которую вы уже читали. Мы начали с изучения основ: функторы, аппликативные функторы и монады. Пошли дальше и увидели монады еще полезнее `Reader`, `Writer` и `State`. Теперь мы изучили как это всё объединять вместе используя преобразователи монад.

Revision #5
Created 2022-03-11 05:45:42 UTC by gasick
Updated 2022-10-09 08:21:21 UTC by gasick