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

Функторы

Добро пожаловать в нашу серию статей. Монады одна из тех идей, которая кажется причиной множества страхов и мучений среди множества людей пробоющих Haskell. Цель этой серии показать, что это не страшная и не сложная идея, и может быть легко разобрана делая определенные шаги.

Простой пример.

Есть простой пример с которого мы начнем наш путь. Этот код превращает входную строку типа `John Doe 24` в кортеж. Мы хотим учитывать все входные варианты, поэтому результатом будет `Maybe`.

```
tupleFromInputString :: String -> Maybe (String, String, Int)
tupleFromInputString input = if length stringComponents /= 3
  then Nothing
  else Just (stringComponents !! 0, stringComponents !! 1, age)
  where
    stringComponents = words input
    age = (read (stringComponents !! 2) :: Int)
```

Эта простая функция принимает строку и преобразует её в параметры для имени, фамилии и возраста. Предположим у нас есть другая часть программы использующая тип данных для отображение человека вместо кортежа. Мы захотим написать функцию преобразователь между этими двумя видами. Мы так же хотим учитывать ситуацию невозможности этого преобразования. Поэтому есть другая функция, которая обработает этот случай.

```
data Person = Person {
  firstName :: String,
  lastName :: String,
```

```
  age :: Int
}

personFromTuple :: (String, String, Int) -> Person
personFromTuple (fName, lName, age) = Person fName lName age

convertTuple :: Maybe (String, String, Int) -> Maybe Person
convertTuple Nothing = Nothing
convertTuple (Just t) = Just (personFromTuple t)
```

Изменение формата

Но, представьте, наша оригинальная программа меняется в части чтения всего списка имен:

```
listFromInputString :: String -> [(String, String, Int)]
listFromInputString contents = mapMaybe tupleFromInputString (lines contents)

tupleFromInputString :: String -> Maybe (String, String, Int)
...
```

Теперь если мы передаем результат коду используя `Person` мы должны изменить тип функции `convertTuple`. Она будет иметь паралельную структуру. `Maybe` и `List` оба действуют как хранитель других значений. Иногда, нас не заботит во что обернуты значения. Нам просто хочется преобразовать что-то лежащее под существующим значением. и затем запустим новое значение в той же обертке.

Введение в функторы

С этой идеи мы можем начать разбирать функторы. Первое и главное: Функтор это класс типа в Haskell. Для типов которые являются экземплярами функторных классов типа, они должны реализовывать простую функцию: `fmap`.

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

Функция `fmap` принимает два ввода. Первый - требует функцию для вдух типов данных. Второй параметр - хранилище первоо типа. Вывод - хранилище второго типа. Теперь взглянем на несколько различных экземпляров функторов для знакомых типов. Для списков, `fmap` просто определяется как базовая функция `map`:

```
instance Functor [] where
  fmap = map
```

На самом деле, `fmap` это обобщение соответствия. Например, тип данных `Map` так же функтор. Он использует свою собственную функцию `map` для `fmap`. Функторы просто берут эту идею преобразования всех ниже лежащих значений и применяют их к другим типам. С этим, давайте взглянем на `Maybe` как на функтор:

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

Выглядит довольно похоже на нашу функцию `convertTuple`. Если у нас нет значения на первом месте, тогда результат `Nothing`. Если имеется значение, тогда просто применяется функция к значение и превращает её в `Just`. Тип данных `Either` может быть типом `Maybe` с дополнительной информацией по какой причине. Он имеет схожее поведение:

```
instance Functor (Either a) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

Отметим, что параметр первого типа этого объекта исправлен. Только второй параметр значения `Either` изменен с помощью `fmap`. Основываясь на этих примерах, мы можем увидеть как переписать `convertTuple` обощеннее:

```
convertTupleFunctor :: Functor f => f (String, String, Int) -> f Person
convertTupleFunctor = fmap personFromTuple
```

Делаем свой функтор

Мы так же можем взять свой собственный тип и определить экмзепляр Функтора. Предположим у нас есть следующий тип данных, отражаюищй папку должностных лиц правительства на местах. Зададим его типом a. Это значит, что мы позволяем различным папкам использовать различные представления должностных лиц.

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

Одна часть нашего приложения может отражать людей с помощь кортежей. Это будет тип `GovDirectory(String, String, Int)`. В то время, как другая часть может использовать тип `GovDirectory Person`. Мы можем определить следующий экземпляр функтора для `GovDirectory` определив `fmap`. Так как наш тип лежащий внутри в вцелом является функтором, это позволяет просто вызывать `fmap` для полей.

```
instance Functor GovDirectory where
  fmap f oldDirectory = GovDirectory {
    mayor = f (mayor oldDirectory),
    interimMayor = fmap f (interimMayor oldDirectory),
    cabinet = fmap f (cabinet oldDirectory),
    councilMembers = fmap f (councilMembers oldDirectory)
  }
```

Так же можно использовать инфиксный оператор `<$>` в качестве синонима `fmap`. Чтобы описать всё гораздо проще:

```
instance Functor GovDirectory where
  fmap f oldDirectory = GovDirectory {
    mayor = f (mayor oldDirectory),
    interimMayor = f <$> interimMayor oldDirectory,
    cabinet = f <$> cabinet oldDirectory,
    councilMembers = f <$> councilMembers oldDirectory
  }
```

Теперь у нас есть свой функтор, преобразование типов данных внутри нашей папки теперь проще. Мы можем просто использовать `fmap` объединив с нашей функцией преобразования, `personFromTuple`:

```
oldDirectory :: GovDirectory (String, String, Int)
oldDirectory = GovDirectory
  ("John", "Doe", 46)
  Nothing
  (M.fromList
    [ ("Treasurer", ("Timothy", "Houston", 51))
    , ("Historian", ("Bill", "Jefferson", 42))
    , ("Sheriff", ("Susan", "Harrison", 49))
    ])
  ([("Sharon", "Stevens", 38), ("Christine", "Washington", 47)])

newDirectory :: GovDirectory Person
newDirectory = personFromTuple <$> oldDirectory
```

Выводы

Теперь вы знаете о функторах, нужно время, чтобы понять эти типы структур. Двигаемся к части 2, где мы обсудим применение функторов.

Revision #6
Created 11 March 2022 05:31:28 by gasick
Updated 5 September 2022 18:45:05 by gasick