

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

Монады

Добро пожаловать в часть 3 нашей серии абстрактных структур! Мы, наконец, коснемся идеи монад! Множество людей пытаются изучить монады без попытки заиметь понимания того, как абстрактные структуры типов класса работают. Это главная причина борьбы. Если вы всё еще этого не понимаете, обратитесь к 1 и 2 части этой серии.

После этой статьи вы будете готовы к тому, чтобы писать свой собственный код Haskell.

Букварь монад

Есть множество инструкций и писаний монад в интернете. Количество аналогий просто смешно. Но вот мои 5 копеек в определении: Монада - обертка значения или вычисления с определенным контекстом. Монада должна определять и смысл обернутого значения в контексте и способ объединения вычислений в контексте.

Это определение достаточно широко. Давайте взглянем на конкретный пример, и попробуем понять.

Классы типы монад

Так же как с функторами и аппликативными функторами, Haskell отражает монады с помощью тип класса. На это есть две функции:

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

Эти две функции отвечают двум идеям выше. Функция возвращения определяем как обернуть значения в контексте монад. Оператор `>>=`, который мы назовем его функцией "связывания", определяет как объединить две операции с контекстом. Давайте проясним это далее изучив несколько определенным экземпляров монад.

Монада Maybe

`Just` как `Maybe` это функтор и аппликативный функтор, но еще и монада. Чтобы понять смысл монады `Maybe` давайте посмотрим представим код:

```
maybeFunc1 :: String -> Maybe Int
maybeFunc1 "" = Nothing
maybeFunc1 str = Just $ length str

maybeFunc2 :: Int -> Maybe Float
maybeFunc2 i = if i `mod` 2 == 0
  then Nothing
  else Just ((fromIntegral i) * 3.14159)

maybeFunc3 :: Float -> Maybe [Int]
maybeFunc3 f = if f > 15.0
  then Nothing
  else Just [floor f, ceiling f]

runMaybeFuncs :: String -> Maybe [Int]
runMaybeFuncs input = case maybeFunc1 input of
  Nothing -> Nothing
  Just i -> case maybeFunc2 i of
    Nothing -> Nothing
    Just f -> maybeFunc3 f
```

Можно увидеть, что мы начинаем разрабатывать отвратительный треугольный шаблон, в качестве продолжения шаблона соответствия результатов успешного вызова функций. Если мы добавили еще больше функций `Maybe` в него, то всё станет еще хуже. Если мы считаем `Maybe` в качестве монады, мы можем сделать код гораздо чище. Давайте взглянем на то, как Haskell реализует `Maybe` монаду, чтобы понять как это делать.

```
instance Monad Maybe where
    return = Just
    Nothing >>= _ = Nothing
    Just a >>= f = f a
```

Внутри `Maybe` монада проста. Вычисления сознанием в `Maybe` могут как пройти, так и не пройти успешно. Мы можем взять любое значение обернуть его в этом контексте вызовом значения `success`. Мы делаем это с помощью конструктора `Just`. Неуспех обозначается с помощью `Nothing`.

Объединим вычисления в контексте проверяя результата первого вычисления. Если успешно, мы берем его значение и передаем во второе вычисление. Если неуспешно, тогда у нас нет значения для передачи дальше. Поэтому результирующее вычисление не будет успешно. Взглянем на то, как мы можем использовать `bind(>>=)` оператор для объединения наших операторов:

```
runMaybeFuncsBind :: String -> Maybe [Int]
runMaybeFuncsBind input = maybeFunc1 input >>= maybeFunc2 >>= maybeFunc3
```

Выглядит гораздо чище! Давайте взглянем почему работают типы. Результат `maybeFunc1` просто `Maybe Int`. Затем оператор `bind(>>=)` позволяет нам взять это `Maybe Int` значение и объединить с `maybeFunc2`, чей тип `Int -> Maybe Float`. Оператор `bind(>>=)` разрешает значение в `Maybe Float`. Затем мы передаем походом образом через оператор `bind(>>=)` в `maybeFunc3` результатом которой является конечный тип: `Maybe [Int]`.

Ваша функции не всегда будут так ясно сочитаться. Тут в силу вступает запись `do`. Код выше можно переписать следующим образом:

```
runMaybeFuncsDo :: String -> Maybe [Int]
runMaybeFuncsDo input = do
    i <- maybeFunc1 input
    f <- maybeFunc2 i
    maybeFunc3 f
```

Оператор `<-` особенный. Он эффективно разворачивает значение с правой стороны монады. Это значит, что значение `i` имеет типа `Int`, даже не смотря на результат `maybeFunc1` как `Maybe Int`. Оператор `bind(>>=)` работает без нашего участия. Если функция возвращает `Nothing`

, тогда вся функция `runMaybeFuncs` вернет `Nothing`.

При беглом осмотре, это выглядит гораздо сложнее, чем пример с `bind(>=>=)`. Однако, оно дает нам гораздо больше гибкости. Предположим, мы хотим добавить 2 к целому числу перед вызовом `MaybeFunc2`. Это проще сделать с помощью `do` записи, но гораздо сложнее используя связывания.

```
runMaybeFuncsDo2 :: String -> Maybe [Int]
runMaybeFuncsDo2 input = do
  i <- maybeFunc1 input
  f <- maybeFunc2 (i + 2)
  maybeFunc3 f

-- Not so nice
runMaybeFuncsBind2 :: String -> Maybe [Int]
runMaybeFuncsBind2 input = maybeFunc1 input
  >>= (\i -> maybeFunc2 (i + 2))
  >>= maybeFunc3
```

Преимущества гораздо очевидны если мы хотим использовать множество прошедших результатов при вызове функции. Используя связывания, мы сможем постоянно складывать аргументы в анонимную функцию.

“ Мы никогда не используем `<-` для развертывания последней операции в блоке `do`.

Наш вызов `maybeFunc3` имеет тип `Maybe [Int]`. Это наш последний тип(не `[Int]`) поэтому его не нужно разворачивать.

монада `Either`

Теперь, давайте посмотрим на монаду `Either`, которая очень похожа на монаду `Maybe`. Вот её определение:

```
instance Monad (Either a) where
  return r = Right r
  (Left l) >>= _ = Left l
  (Right r) >>= f = f r
```

Поскольку `Maybe` имеет успех или не успех со значением, монада `Either` прикладывает информацию к неудаче. `Just` как `Maybe` оборачивает значение в его контексте вызова делая его успешным. Монадическое поведение так же объединяет операции завершаясь на первом не успехе. Давайте посмотрим как мы можем использовать это чтобы сделать наш код чище.

```
eitherFunc1 :: String -> Either String Int
eitherFunc1 "" = Left "String cannot be empty!"
eitherFunc1 str = Right $ length str
```

```
eitherFunc2 :: Int -> Either String Float
eitherFunc2 i = if i `mod` 2 == 0
  then Left "Length cannot be even!"
  else Right ((fromIntegral i) * 3.14159)
```

```
eitherFunc3 :: Float -> Either String [Int]
eitherFunc3 f = if f > 15.0
  then Left "Float is too large!"
  else Right [floor f, ceiling f]
```

```
runEitherFuncs :: String -> Either String [Int]
runEitherFuncs input = do
  i <- eitherFunc1 input
  f <- eitherFunc2 i
  eitherFunc3 f
```

Любой не успех просто даст нам значение `Nothing`:

```
>> runMaybeFuncs ""
Nothing
>> runMaybeFuncs "Hi"
Nothing
>> runMaybeFuncs "Hithere"
Nothing
```

```
>> runMaybeFuncs "Hit"
Just [9,10]
```

когда мы запустим наш код, мы можем посмотреть на строковый результат ошибки, и она расскажет нам о том, какая функция не смогла произвести вычисления.

```
>> runMaybeFuncs ""
Left "String cannot be empty!"
>> runMaybeFuncs "Hi"
Left "Length cannot be even!"
>> runMaybeFuncs "Hithere"
Left "Float is too large!"
>> runMaybeFuncs "Hit"
Right [9,10]
```

Заметим, что мы параметризовали монаду `Either` с помощью нашего типа ошибки. Если у нас есть:

```
data CustomError = CustomError

maybeFunc2 :: Either CustomError Float
...
```

Это функция теперь новая монада. Объединения с другими функциями не будет легким.

Монада IO

Монада IO, возможно, самая важная монада в Haskell. Это так же одна из самых сложных монад для понимания начинающих. Её реализация достаточно сложна для обсуждения при первом знакомстве с языком. Поэтому будем учиться по примерам.

IO монада обортывает вычисления с ледующем случае: "Вычисления могут читать информацию или писать в терминал, файловую систему, ОС или сеть". Если выхотите получить пользовательский ввод, выведите сообщение пользователю, прочитайте информацию из файла, или сделайте сетевой вызов, для этого понадобится IO монада. Эти вызовы имеют "сторонние эффекты", мы нне может произвести их из "чистого" Haskell кода.

Важная работа почти любого компьютера это взаимодействие с внешним миром, каким-то образом. На этот случай, корнем всего выполняемого Haskell кода это функция называемая `main`, с типом `IO()`. Поэтому любая программа начинается с `IO` монады. Отсюда вы можете получить любой необходимый ввод, вызвать относительно "чистый" код с помощью ввода, и затем вывести результат каким-то образом. Обратное не работает. Вы не можете вызывать внутри `IO` кода, код, как тот, который вы можете вызвать в `Maybe` функции из чистого кода.

Давайте взглянем на простой пример показывающий несколько базовых `IO` функций. Мы будем использовать `do`-запись для того, чтобы показать схожесть с другими монадами, которые мы уже встречали. Выведем тип каждой `IO` функции для ясности.

```
main :: IO ()
main = do
  -- getLine :: IO String
  input <- getLine
  let uppercased = map Data.Char.toUpper input
  -- print :: String -> IO ()
  print uppercased
```

Каждый раз мы видим строку нашей программы и она имеет тип `IO`. Так же как мы можем развернуть `i` в примере `maybe` для получения `Int` взамен `Maybe Int`, мы можем использовать `<-`, чтобы развернуть результат `getLine` в качестве `String`. Мы можем затем использовать это значение с помощью строковой функции, и передать результат в функцию `print`.

Это просто эхо-программа. Она читает строку из терминала и затем выводит строку обратно с капсом. Надеюсь она дает вам базовое понимание того как `IO` работает. Мы залезем глубже в детали в следующей паре статей.

Выводы

С этой точки, мы должны наконец иметь лучшее понимание того, что такое монады. Но если они не имеют смысла до сих пор, не раздражайтесь! Мне пришлось потратить несколько попыток, прежде чем я смог понять их. Не бойтесь взглянуть еще разок на 1 и 2 части, чтобы освежить Haskell знания. И определенно стоит прочитать еще разок эту статью.

Если же вам всё понятно, вы готовы двигаться к части 4, где вы изучите о [Reader](#) и [Writer](#) монадах, что позволит вам привнести возможность использовать некий функционал в Haskell, о котором вы думали, что он не доступен.

If you've never programmed in Haskell before, hopefully I've convinced you that it's not that scary and you're ready to check it out! Download our [Beginners Checklist](#) to learn how to get started.

Revision #5

Created 11 March 2022 05:36:49 by gasick

Updated 26 September 2022 13:06:05 by gasick