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

Монады (и другие функциональные структуры)

Монады одни из самых страшных тем для новичков в Haskell. Но как и большинство идей, они перестают быть страшными с приходом понимания. Есть множество пособий и множество определений, что такое на самом деле монады и как это объяснить людям. Эта серия - попытка облегчить это больное место для тех кто начинает изучать Haskell. Мы будем делать всё с базовых понятий, начнем с функторов и аппликативных функторов чтобы иметь представление как абстрактные структуры работают в Haskell. Затем потрогаем монады и взглянем на самые распространенные.

- Функторы
- Аппликативные функторы
- Монады
- Монады Reader и Writer
- State Монада
- Преобразователи Монад
- Законы Монад

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.

Аппликативные функторы

Добро пожаловать во вторую часть серии о монадах и других функциональных структур. Мы продолжим готовить собирать нашу базу изучая идеи апликативных 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 исопльзовать `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 функци обернутую в `Maybe`. Но мы до сих пор не может 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 апликативные типы классов, говорят две главные функции:

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

Чистая фнукция принимает какое-то значение и обертывает его в минимальный контекст. Функция `<*>` вызывает последующиее приложение, которое принимает 2 параметра. Первый - принимает функци обернутую в конекст. 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 ычасто 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 начение с помощью `Just`. Затем связывает веще вмете, если другие функции или знаения будут `Nothing`, мы просто выводим `Nothing`. В противном случае применяем функцию к значение 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 олько одна функция, этот резаультат имеет понятноее 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 паралельное применение функторов. Например, мы можемт хотеть использовать второй список из примера выше, но иметь результат `[2,10,30]`. Для этого есть конструкт под названием `Ziplist`, это новый тип вокрут списка, для которого поведение экземпляр аппликатива и предусмотренно.

```
>> 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 гораздно проще понять если попытаться исполнить код из примеров самостоятельно.

Монады

Добро пожаловать в часть 3 нашей серии абстрактных структур! Мы, наконец, коснемся идеи монад! Множежство 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 пониматете, 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 поисаний 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 достатоно 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 узучив 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 созначением 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 результа 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 исопльзовать `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 походим образом через оператор `bind(>>=)` в `maybeFunc3` результатом которой является конечный тип: `Maybe [Int]`.

Ваша функции не всегд 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 сложноее 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 функии. Используя связывания, мы сможем постоянно складывать аргументы в анонимную функцию.

> Мы никогда не используем `<-` для развернывания 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 неуспеху. `Just` как `Maybe` обертывает значение в его контексте вызова делая его упешным. Монадическое поведение так же объединяет операции завершаясь на первом не успехе. Давайте посмотрим как мы можем использовать это чтобы сделать наш код выше 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, возмоно, самая важная монада в Haskell. Это так же одна из самых сложных монад для понимания начинающих. Её реализация достаточна сложна для обсуждения при первом знакомстве с языком. Поэтому будем учиться по примерам.

IO монада обертывает 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 понадобиться 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 взывать 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 передаавть 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.

Монады Reader и Writer

В части 3 этой серии, мы наконец затронули идею монад. Мы изучили что они такое, и увидели как некоторые общие типы, например `IO` и `Maybe`, работают в качестве монад. В этой части, мы посмотрим на некоторые другие полезные монады. В частности мы рассмотрим монады `Reader` и `Writer`.

Глобальные переменные(или их нехватка)

В Haskell, наш код в общем "чистый", что значит, что функции могут только взаимодействовать с аргументами переданными им. Смысл в том, чтобы мы не могли имметь глобальных переменных. Мы можем ипеть глобальные выражения, но они фиксируются во время компиляции. Если поведение пользователя может изменить их, нам нужно обернуть их в `IO` монаду, что значит, что мы не можем использовать её в "чистом" коде.

Представим следующий пример. Мы хотим иметь `Environment` содержащее параметры в качестве глобальных переменных. Однако, мы должны их загрузить через конфигурационный файл или командную строку, что трубует `IO` монаду.

```
main1 :: IO ()
main1 = do
  env <- loadEnv
  let str = func1 env
  print str

data Environment = Environment
```

```
  { param1 :: String
  , param2 :: String
  , param3 :: String
  }

loadEnv :: IO Environment
loadEnv = ...

func1 :: Environment -> String
func1 env = "Result: " ++ (show (func2 env))

func2 :: Environment -> Int
func2 env = 2 + floor (func3 env)

func3 :: Environment -> Float
func3 env = (fromIntegral $ l1 + l2 + l3) * 2.1
  where
    l1 = length (param1 env)
    l2 = length (param2 env) * 2
    l3 = length (param3 env) * 3
```

Функция на самом деле используется `func3`. Однако `func3` чистая функцияю. Это значит, она не может вызывать напрямую `loadenv`, так как она не "чистая" функция. Это значит, что окружение должно быть передано через переменную в другую функцию, чтобы можно было передать её в функцию `func3`. В языке с глобальными переменными, мы должны сохранить `env` в качестве глобальной переменой в `main`. Функция `func3` должна иметь доступ напрямую. Не нужно иметь парметра для `func1` и `func2`. В больших программах эта передача переменных может устроить головную боль.

Решение READER

Монада `Reader` решает эту проблему. Она создает глобальное только для чтения значение определенного типа. Все функции внутри монады могут прочитать "тип". Давайте взглянем на то как монада `Reader` меняет форму нашего кода. Наши функции больше не трубуют `Environment`в качесте обязательного параметра, так как они могут получить доступ к ней через монаду.

```
main :: IO ()
main = do
  env <- loadEnv
  let str = runReader func1' env
  print str

func1' :: Reader Environment String
func1' = do
  res <- func2'
  return ("Result: " ++ show res)

func2' :: Reader Environment Int
func2' = do
  env <- ask
  let res3 = func3 env
  return (2 + floor res3)

-- as above
func3 :: Environment -> Float
...
```

Функция `ask` развертывает окружение для того, чтобы мы могли его исопльзовать. Привязывание действий к моанадам позволяет нам связать различные `Reader` действия. Для того, чтобы вызвать действие чтения из чистого кода, нужно вызвать `runReader` функцию и подать окружение в качестве параметра. Все функции внутри действия будут обращаться как к глобальной переменной.

Код выше так же вводит важное понятие. Каждый раз, когда вы вводите понятие монада "X", всегда есть соответстующая функция "runX", которая говорит вам как запустить операции над монадой из чистого контекста(IO исключение). Эта функция будет часто требоваться при определенном вводе, так же как и сами вычисления. Затем оно будет производить вывод этим самых вычислений. В этом случае `Reader`, у нас есть `runReader` функция. Она требует значение, которое мы будем читать, и сами вычисления `Reader`.

```
runReader :: Reader r a -> r -> a
```

Может быть не похоже, что нам многое удалось, но наш код более понятен теперь. Мы сохранили `func3`, так как она есть. Она имеет смысл, чтобы описать её в качестве

переменной из `Environment` с помощью функции. Однако, наши другие две функции больше не принимают окружение как обязательные параметры. Они просто существуют в контексте где окружение - глобальная переменная.

Сбор значений

Чтобы понять монаду `Winter`, давайте поговорим о проблеме сбора. Предположим у нас есть несколько различных функций. Каждая делает строковые операции, которые чего-то стоят. Мы хотим отслеживать сколько "стоят" все вычисления вместе. Мы можем сделать следующее, для сбора аргументов и слежения за "ценой" которую мы получим. Мы продолжаем передавать собранные переменные вместе с результатом обработки строки.

```
-- Calls func2 if even length, func3 and func4 if odd
func1 :: String -> (Int, String)
func1 input = if length input `mod` 2 == 0
  then func2 (0, input)
  else (i1 + i2, str1 ++ str2)
    where
      (i1, str1) = func3 (0, tail input)
      (i2, str2) = func4 (0, take 1 input)

-- Calls func4 on truncated version
func2 :: (Int, String) -> (Int, String)
func2 (prev, input) = if (length input) > 10
  then func4 (prev + 1, take 9 input)
  else (10, input)

-- Calls func2 on expanded version if a multiple of 3
func3 :: (Int, String) -> (Int, String)
func3 (prev, input) = if (length input) `mod` 3 == 0
  then (prev + f2resI + 3, f2resStr)
  else (prev + 1, tail input)
  where
    (f2resI, f2resStr) = func2 (prev, input ++ "ab")

func4 :: (Int, String) -> (Int, String)
func4 (prev, input) = if (length input) < 10
```

```
  then (prev + length input, input ++ input)
  else (prev + 5, take 5 input)
```

Для начала, можно отметить. что структура функции несколько трудно обслуживаемая. Опять, мы передаем дополнительные параметры. В частности, мы отслеживаем общую стоимость, которая показывается для ввода и вывода каждой функции. Монада `Writer` дает нам простой способ отслеживания значений. Она так же делает легче ля нас отображение стоиомсти для различных типов. Но чтобы понять, как мы должны для начала изучить два типокласса, `Semigroup` и `Monoid`, которые помогут обощить сбор.

SEMIGROUPS и MONOIDS

`Semigroup` это любой тип, который мы собираем с помощью "append" оператора. Эта функция использует оператор `<>`. Она объединяет два элемента типа в новый, третий.

```
class Semigroup a where
  (<>) :: a -> a -> a
```

Для нашего первого простого примера, мы можем думать представить `Int` тип как часть `Semigroup` под операцией сложнения.

```
instance Semigroup Int where
  a <> b = a + b
```

`Monoid` расширяет определение `Semigroup`, чтобы можно было включить определяющий элемент. Этот элемент называется `mempty`, так как это "empty" элементо сортировки. Отметим, что ограничение `Monoid` в том, что он уже должен быть `Semigroup`.

```
class (Semigroup a) => Monoid a where
  mempty :: a
```

Определяющий элемент долен иметь свойства, если мы прибавшяем любой другой элемент `a`, в любом направлении, результатом должен быть `a`. Поэтому результатом `a <> mempty == a` и `mempty <> a == a` всегда должны быть `true`. Мы можем расширить наше определение `Int` для `Semigroup` добавив `0` в качестве определяющего элемента для `Monoid`.

```
instance Monoid Int where
  memty = 0
```

Мы можем продуктивно использовать `Int` и собирать класс. Функция `mempty` предлагает начальное значение для нешего моноида. Затем с помощью `mappend`, мы объединяем два значения этого типа в результат. Это довольно легко, сделать экземпляр `Monoid` для `Int`. Наш счетчик начинается с `0`, и мы можем объединить значения для добавления.

Этот `Int` экземпляр не доступен по умолчания. Это потому, что мы может так же предоставить `Monoid` из `Int` используя перемножение вместо сложения. В этом случае, 1 становится определяющим.

```
instance Semigroup Int where
  a <> b = a * b

instance Monoid Int where
  mempty = 1
```

В обоих случаях `Int` пример, наша `append` функция суммирующая. Базовая библиотека включет экземпляр `Monoid` для любого типа `List`. Оператор `append` использует оператор прибавления списка `++`, который не суммирующий. В этом случае определяющий элемент это пустой список.

```
instance Semigroup [a] where
  xs <> ys = xs ++ ys

instance Monoid [a] where
  mempty = []

-- Not commutative!
-- [1, 2] <> [3, 4] == [1, 2, 3, 4]
-- [3, 4] <> [1, 2] == [3, 4, 1, 2]
```

Использование WRITER для отслеживания ACCUMULATOR

Как же это помогает нам с проблемой сложения выше?

Монада `Writer` параметризуется с помощью некоторого моноидного типа. Его задача следить за складываемым значением этого типа. Его цель жить в контексте глобальной переменной которую они могут менять. Пока `Reader` дает нам возможность читать глобальную переменную, но не менять её `Writer` позволяет нам менять значение с помощью сложения, при этом нельзя её читать при вычислении. Мы можем вызвать операцию добавления используя `tell` функцию в цели нашего выражения `Writer`.

```
tell :: a -> Writer a ()
```

Так же как и с `Reader` и `runReader`, есть `runWriter` фукнция. И выглядит она немного по другому.

```
runWriter :: Writer w a -> (a, w)
```

Нам не нужно предоставлять дополнительный ввод кроме вычислений для запуска. Но `runWriter` осуществляет 2 вывода! Первый это результат нашего вычисления. Второй - последнее сложенное значение для `writer`. Мы не предоставили входного значения, так как он автоматически использует `mempty` из `Monoid`!

Давайте изучим как изменить наш код выше, чтобы использовать эту монаду. начнем с `acc2`

.

```
acc2' :: String -> Writer Int String
acc2' input = if (length input) > 10
  then do
    tell 1
```

```
    acc4' (take 9 input)
  else do
    tell 10
    return input
```

Создаем отдельную ветку по количествую ввходных данных, и для кадой ветки выполняем `do`. Будем использовать `tell` для предоставления соответствующего значения для увеличивания сумматора, и затем двигается к вызову следующей функции, или возвращаем ответ. затем `acc3` и `acc4`.

```
acc3' :: String -> Writer Int String
acc3' input = if (length input) `mod` 3 == 0
  then do
    tell 3
    acc2' (input ++ "ab")
  else do
    tell 1
    return $ tail input

acc4' :: String -> Writer Int String
acc4' input = if (length input) < 10
  then do
    tell (length input)
    return (input ++ input)
  else do
    tell 5
    return (take 5 input)
```

Наконец, мы не меняем тип подписи нашей оригинальной функци, вместо этого мы используем `runWriter` для вызова помощника, как и положено.

```
acc1' :: String -> (String, Int)
acc1' input = if length input `mod` 2 == 0
  then runWriter (acc2' input)
  else runWriter $ do
    str1 <- acc3' (tail input)
    str2 <- acc4' (take 1 input)
    return (str1 ++ str2)
```

Отемтим, нам больше не нужно явно отслеживать сумматор. Он не обернут с помощью `writer` монады. Мы можем увеличить его в любой нашей функии вызвав `tell`. Теперь наш код граздо проще а типы яснее.

Выводы

Теперь, зная про `Reader` и `Writer` монады, пришло время двигаться дальше. Дальше мы обсудим монаду `State`. Эта монада объединяет эти две идеи в `read/write state`, естественно позволяя использовать глобальные переменные на полную. Если эти идеи до сих пор вас смущают, не бойтесь перечитать статью.

State Монада

В прошлой части, мы изучили монады `Reader` и `Writer`. Они пакаказил, что на самом деле имеем алтернативу глобальным переменным. Нам просто нужно каким-то образом заключить их в определенный тип, это то для чего они нужны. В этой части изучим `State` монаду, которая объединяет некоторую функциональность для обоих идей.

Мотивации пост: Крестики-нолики

Для этой части мы воспользуемся простой моделью для игры Крестки-нолики. Главный объект это тип данных `GameState` содержащий несколько важных кусочков информации. Первое и важное, он содержит "доску", и двумерный массив индексов состояния полей(X/0 или пусто). Так же знает чей ход и имеет случайный генератор.

```
data GameState = GameState
  { board :: A.Array TileIndex TileState
  , currentPlayer :: Player
  , generator :: StdGen
  }

data Player = XPlayer | OPlayer

data TileState = Empty | HasX | HasO
  deriving Eq

type TileIndex = (Int, Int)
```

Давай взглянем на то, как некоторые из функций нашей игры будут работать. Например нужно придумать функцию для случайного выбора хода. Она долна выводить `TileIndex` и изменять генератор нашей игры. Затем основываясь на нем делаем шаг и передаем ход

другому игроку. Другими словами, у нас есть операции которые зависят от текущего состояния игры, но так же обновляет это состояние.

THE STATE MONAD

This is exactly the situation the State monad deals with. The State monad wraps computations in the context of reading and modifying a global state object. This context chains two operations together in an intuitive way. First, it determines what the state should be after the first operation. Then, it resolves the second operation with the new state.

It is parameterized by a single type parameter s, the state type in use. So just like the Reader has a single type we read from, the State has a single type we can both read from and write to. There are two primary actions we can take within the State monad: get and put. The first retrieves the state, the second modifies it by replacing it with a new object. Typically though, this new object will be similar to the original:

```
-- Retrieves the state, like Reader.ask
get :: State s s

-- Overwrites the existing state
put :: s -> State s ()
```

There is also a runState function, similar to runReader and runWriter. Like the Reader monad, we must provide an initial state, in addition to the computation to run. But then like the writer, it produces two outputs: the result of our computation AND the final state:

```
runState :: s -> State s a -> (a, s)
```

If we wish to discard either the final state or the computation's result, we can use evalState and execState, respectively:

```
evalState :: State s a -> s -> a

execState :: State s a -> s -> s
```

So for our Tic Tac Toe game, many of our functions will have a signature like State GameState a.

OUR STATEFUL FUNCTIONS

Now we can examine some of the different functions mentioned above and determine their types. We have for instance, picking a random move:

```
chooseRandomMove :: State GameState TileIndex
chooseRandomMove = do
  game <- get
  let openSpots = [ fst pair | pair <- A.assocs (board game), snd pair == Empty]
  let gen = generator game
  let (i, gen') = randomR (0, length openSpots - 1) gen
  put $ game { generator = gen' }
  return $ openSpots !! i
```

This outputs a TileIndex to us, and modifies the random number generator stored in our state! Now we also have the function applying a move:

```
applyMove :: TileIndex -> State GameState ()
applyMove i = do
  game <- get
  let p = currentPlayer game
  let newBoard = board game A.// [(i, tileForPlayer p)]
  put $ game { currentPlayer = nextPlayer p, board = newBoard }

nextPlayer :: Player -> Player
nextPlayer XPlayer = OPlayer
nextPlayer OPlayer = XPlayer

tileForPlayer :: Player -> TileState
tileForPlayer XPlayer = HasX
tileForPlayer OPlayer = HasO
```

This updates the board with the new tile, and then changes the current player, providing no output.

So finally, we can combine these functions together with do-syntax, and it actually looks quite clean! We don't need to worry about the side effects. The different monadic functions handle them. Here's a sample of what your function might look like to play one turn of the game. At the end, it

returns a boolean determining if we've filled all the spaces:

```
resolveTurn :: State GameState Bool
resolveTurn = do
  i <- chooseRandomMove
  applyMove i
  isGameDone

isGameDone :: State GameState Bool
isGameDone = do
  game <- get
  let openSpots = [ fst pair | pair <- A.assocs (board game), snd pair == Empty]
  return $ length openSpots == 0
```

Obviously, there are some more complications for how the game would work in full, but the general idea should be clear. Any additional functions could live within the State monad.

STATE, IO, AND OTHER LANGUAGES

When thinking about Haskell, it is often seen as a restriction that we can't have global variables like you could with Java class variables. However, we see now this isn't true. We could have a data type with exactly the same functionality as a Java class. We would just have many functions that can modify the global state of the class object using the State monad.

The difference is in Haskell we simply put a label on these types of functions. We don't allow it to happen for free. We want to know when side effects can potentially happen, because knowing when they can happen makes our code easier to reason about. In a Java class, many of the methods won't actually need to modify the state. But they could, which makes it harder to debug them. In Haskell we can simply make these pure functions, and our code will be simpler.

IO is the same way. It's not like we can't perform IO in Haskell. Instead, we want to label the areas where we can, to increase our certainty about the areas where we don't need to. When we know part of our code cannot communicate with the outside world, we can be far more certain of its

behavior.

SUMMARY

That wraps it up for the State monad! Now that we know all these different monad constructs, you might be wondering how we can combine them. What if there was some part of our state that we wanted to be able to modify (using the State monad), but then there was another part that was read-only. How can we get multiple monadic capabilities at the same time? To learn to answer, head to part 6! In the penultimate section of this series, we'll discuss monad transformers. This concept will allow us to compose several monads together into a single monad!

Now that you're starting to understand monads, you can really pick up some steam on learning some useful libraries for important tasks. Download our Production Checklist for some examples of libraries that you can learn!

Преобразователи Монад

В нескольких прошлых частях серии, мы изучили множество новых монад. В 3 части мы увидели как часто вещи как `Maybe` и `IO` могут быть монадами. Затем в 4 и 5 частях мы изучили Reader, Writer и State монады. С этими монадами на поясе, вы возмоно думаете как можно их объединять. Ответ, как мы обнаружи в этой части, это преобразователи монад.

С пониманием монад, вы открываете больше Haskell возможностей. Но вам всё ещё нужны идеи библиотек Haskell, который позволят вам их испытать.

Пример Мотивации

Ранее, мы уже видели как монада `maybe` помогает избежать треугольника судьбы шаблонов кода. Без них, нам нужно проверять каждую функцию на успех. Однако, примеры на которые мы смотрим, где всё является чистым кодом предполагает следующее:

```
main1 :: IO ()
main1 = do
  maybeUserName <- readUserName
  case maybeUserName of
    Nothing -> print "Invalid user name!"
    Just (uName) -> do
      maybeEmail <- readEmail
      case maybeEmail of
        Nothing -> print "Invalid email!"
        Just (email) -> do
          maybePassword <- readPassword
          Case maybePassword of
            Nothing -> print "Invalid Password"
            Just password -> login uName email password

readUserName :: IO (Maybe String)
readUserName = do
  putStrLn "Please enter your username!"
```

```
  str <- getLine
  if length str > 5
    then return $ Just str
    else return Nothing

readEmail :: IO (Maybe String)
readEmail = do
  putStrLn "Please enter your email!"
  str <- getLine
  if '@' `elem` str && '.' `elem` str
    then return $ Just str
    else return Nothing

readPassword :: IO (Maybe String)
readPassword = do
  putStrLn "Please enter your Password!"
  str <- getLine
  if length str < 8 || null (filter isUpper str) || null (filter isLower str)
    then return Nothing
    else return $ Just str

login :: String -> String -> String -> IO ()
...
```

В этом примере, все наши потенциальные проблемы кода идут из `IO` монады. Как мы може использовать `Maybe` монаду когда мы уже внутри другой монады?

Преобразователи Монад

К счастью, мы можем получить желаемое поведение используя преобразователи монад для объединения. В этом примере, мы обернем `IO` действиее внутрь преобразованной монады `MaybeT`.

Преобразователи Монад это оберточный тип. В общем параметризируемый другим монадическим типом. Затем вы можете запустить действие из внутренней монады, в то время пока добавляете ваше собственное поведение для действия объединения в новую монаду. Общий преобразователь добавляет `T` в конец существующей монады. Ниже

представленно определение `MaybeT`:

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance (Monad m) => Monad (MaybeT m) where
  return = lift . return
  x >>= f = MaybeT $ do
    v <- runMaybeT x
    case v of
      Nothing -> return Nothing
      Just y  -> runMaybeT (f y)
```

`MaybeT` сам по себе это newtype. Он содержит обертку над значением `Maybe`. Если тип `m` это `monad`, мы можем так же сделать монаду из `MaybeT`.

Представим наш пример. Мы хотим использовать `MaybeT` для оборачивания `IO` монады, чтобы запустить `IO` действия. Это значит, что наша новая монада `MaybeT IO`. Наши три вспомогательные функции все возвращают строки, поэтому каждая из них получает тип `MaybeT IO String`. Для преобразования старого `IO` кода в `MaybeT` монаду, всё, что нужно - обернуть `IO` действие в `MaybeT` конструктор.

```
readUserName' :: MaybeT IO String
readUserName' = MaybeT $ do
  putStrLn "Please enter your Username!"
  str <- getLine
  if length str > 5
    then return $ Just str
    else return Nothing

readEmail' :: MaybeT IO String
readEmail' = MaybeT $ do
  putStrLn "Please enter your Email!"
  str <- getLine
  if length str > 5
    then return $ Just str
    else return Nothing

readPassword' :: MaybeT IO String
readPassword' = MaybeT $ do
```

```
  putStrLn "Please enter your Password!"
  str <- getLine
  if length str < 8 || null (filter isUpper str) || null (filter isLower str)
    then return Nothing
    else return $ Just str
```

Теперь ы можем обернуть все три этих вызова в одно монадическое действие, и сделать простое сравнение для получения результата. Мы воспользуемся `runMaybeT` функцией для развертывания значения `Maybe` из `MaybeT`:

```
main2 :: IO ()
main2 = do
  maybeCreds <- runMaybeT $ do
    usr <- readUserName
    email <- readEmail
    pass <- readPassword
    return (usr, email, pass)
  case maybeCreds of
    Nothing -> print "Couldn't login!"
    Just (u, e, p) -> login u e p
```

И этот новый код бдует иметь правильное простое поведение для `Maybe` монады. Если какая-то функция `read` упадет, наш код сразу же вернет `Nothing`.

Добавление уровней.

Вот и мы дождались долгожданное части о преобразователях монад. Так как наш новосозданный тип сам по себе монада, мы межем обернуть её внутри другого преборазователя. Почти все распространненые монады имеют преобразователь типа, `MaybeT` в том числе, это преобразователь для обычной `Maybe` монады.

Для быстрого примера, предположим, у нас есть `Env` тип содержащий пользовательскую информацию. Мы можем обернуть это окружение в `Reader`. Однако, мы хоти всё еще иметь доступ к `IO` функциональности, поэтму мы воспользумся `Reader` преобразователем. Затем обернем резултат с помощью `MaybeT`.

```
type Env = (Maybe String, Maybe String, Maybe String)

readUserName'' :: MaybeT (ReaderT Env IO) String
readUserName'' = MaybeT $ do
  (maybeOldUser, _, _) <- ask
  case maybeOldUser of
    Just str -> return $ Just str
    Nothing -> do
      -- lift allows normal IO functions from inside ReaderT Env IO!
      lift $ putStrLn "Please enter your Username!"
      input <- lift getLine
      if length input > 5
        then return (Just input)
        else return Nothing
```

Заметим, что у нам нужно использовать `lift` для запуска `IO` функции `getLine`. В преобразователе монады, `lift` функция позволяет нам запустить действия нижележащей монады. Это поведение захватывается классом `MonadTrans`:

```
class MonadTrans t where
  lift :: (Monad m) => m a -> t m a
```

Использование `lift` в `ReaderT Env IO` действии позвляет `IO` функцияю Использование типа шаблона из класса, мы можем заменить `Reader Env` на `t` и `IO` на `m`.

Внутри `MaybeT (ReaderT Env IO)` функции, вызываемой `lift` позволяет вам запустить функцию `Reader`. Нам не нужно то что выше, так как набор кода лежит в `Reader` действии в обертке `MaybeT` конструктора.

Чтобы понять идею лифтинга, подумайте о уровне вашей монады как о стеке. Когда вы имеете `ReaderT Env IO` действие, представьте, что `Reader Env` монада сверху `IO` монады. `IO` действие лежит на нижнем уровне. Поэтому, чтобы запустить всё это дело с верхнего слоя, вам нужно сначала подняться. Если ваш стек имеет больше чем 2 слоя, вы можете подниматься несколько раз. Вызывая дважды `MaybeT (ReaderT Env IO)` монаду позволит вам вызывать `IO` функцию.

Не удобно каждый раз знать сколько раз тебе нужно вызывать функцию `lift` для получения текущего уровня. Отсюда вспомогательная функкйия часто используется для этого.

Вдобавок, после преобразования монады, можно запустить несколько уровней, типы могут становится сложнее. Поэтому обычно используют библиотеку `synonyms`.

```
type TripleMonad a = MaybeT (ReaderT Env IO) a

performReader :: ReaderT Env IO a -> TripleMonad a
performReader = lift

performIO :: IO a -> TripleMonad a
performIO = lift . lift
```

Типоклассы

В качестве похожей идеи, есть `typeclass` который позволяет нам сделать определенные предположения о стеке монады. Для примера, вас часто не волнует, что именно в стеке, но вам нужен `IO` где-то внутри. В этом и заключается цель использования `MondaIO` типокласса.

```
class (Monad m) => MonadIO m where
  liftIO :: IO a -> m a
```

We can use this behavior to get a function to print even when we don't know its exact monad:

```
debugFunc :: (MonadIO m) => String -> m ()
debugFunc input = liftIO $ putStrLn ("Successfully produced input: " ++ input)
```

Даже не смотря на то, что функция явно не находится в `MaybeT IO`, мы можем написать нашу версию `main` функции чтобы использовать её.

```
main3 :: IO ()
main3 = do
  maybeCreds <- runMaybeT $ do
    usr <- readUserName'
    debugFunc usr
    email <- readEmail'
    debugFunc email
    pass <- readPassword'
    debugFunc pass
```

```
    return (usr, email, pass)
  case maybeCreds of
    Nothing -> print "Couldn't login!"
    Just (u, e, p) -> login u e p
```

> “ Вы не можете, в общем, обернуть другую монаду с помощью `IO` монады используя преобразователь. Однако, можно сделать другое монадическое значение чтобы вренуть тип `IO` действия.

```
func :: IO (Maybe String)
-- This type makes sense

func2 :: IO_T (ReaderT Env (Maybe)) string
-- This does not exist
```

Выводы

Теперь, вы знаете, как объединять ваши монады, вы почти завершили понинмание ключевых идей! Вы, возможно, хотите попробовать начать писать достаточно сложный код. Но, чтобы научиться владеть монадами, вам нужно знать как делать свою собственную монаду, и для этого вам нужно понять последню идею. Это идея типа `laws`. Каждая структура, которую мы прошли в этой части лекций, связана с `laws`. И чтобы ваши примеры имели смысл, они должны следовать `laws`(т.е. закону). Проверьте 7 главу, чтобы понять, понимаете ли вы что происходит.

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.