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

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.

Revision #3
Created 2022-03-11 05:39:08 UTC by gasick
Updated 2022-09-27 20:55:44 UTC by gasick