Eduardo Ribeiro de Oliveira 11201810478

Introdução às Mônadas em Haskell

Complicadas para alguns, simples para outros, as mônadas sempre geram debates na comunidade de Haskell devido à dificuldade geral para entender o conceito por trás dessa typeclass.
Como você já está tentando entender algo tão complexo, vou partir do princípio que você já entenda o básico de funções em Haskell. Portanto, partiremos de algo um pouco mais avançado que simples funções, vamos relembrar como *Functors* funcionam.
Os Functors são usados quando se tem um valor dentro de um contexto, mas não podemos simplesmente aplicar uma função a esse valor, logo criamos uma instância de Functor para definir como aplicamos funções a valores nesse determinado contexto, e então junto do *fmap* conseguimos aplicar funções a valores num contexto mantendo o mesmo.
Ademais, temos os *Applicatives*, para podermos aplicar funções que tenham um contexto em valores que também tenham contexto e manter o contexto mesmo depois de aplicar essas funções.
Porém como fazemos para pegar um valor que está dentro de um contexto, uma função que recebe um valor fora de contexto mas retorna um valor dentro de contexto? Um pouco confuso né, mas vamos lá.

Conceito

Antes de tudo, vamos ver como a classe Monad é definida em Haskell:

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

Começando pelo return, percebe-se que é uma função simples, ela pega um valor qualquer e simplesmente o coloca dentro de um contexto.
Agora a próxima função é definida como >>= (leia-se bind), essa já é um pouco mais complicada, vamos por partes. O primeiro argumento é um valor do tipo m a em que a pode ser de qualquer tipo, porém m precisa de uma mônada. O segundo é uma função

`a -> m b,` perceba que essa função recebe o tipo a porém fora do contexto m e então devolve um tipo b qualquer dentro do contexto m. Por fim, a função devolve b dentro do contexto m. Para abstrair um pouco, vamos a um exemplo:

Vamos usar o Maybe, que é uma mônada e já está implementada da seguinte forma:

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

Nada muito diferente, né? A única diferença da definição é que quando o argumento é Nothing, o retorno sempre será Nothing, já que não importa o que a função faça nada sempre será nada. Então seguindo em frente, vamos criar uma função simples utilizando o Maybe, no qual ela recebe um Int x e devolve um Just Int (x+10) caso x >= 10 ou Nothing caso contrário. A implementação ficaria desse jeito:

```
soma10 :: Int -> Maybe Int
soma10 x
   | x >= 10 = Just (x + 10)
   | otherwise = Nothing
```

Agora, vamos testá-la para ver se está funcionando corretamente:

```
ghci> soma10 50
Just 60
ghci> soma10 (-5)
Nothing
```

Tudo certo! Bom, agora temos uma função que recebe um valor Int fora de contexto e devolve um valor Int dentro de um contexto, ou seja, podemos utilizar o bind para entendê-lo melhor. Segue alguns exemplos:

```
ghci> Just 20 >>= soma10
Just 30
ghci> Just (-20) >>= soma10
Nothing
ghci> Nothing >>= soma10
Nothing
```

Perceba a função do bind no primeiro exemplo, ela pega um valor que está num contexto, nesse caso o Just 20, extraí o valor do contexto, e aplica a função soma10 nesse valor. Legal, né? Agora você pode se perguntar, e se eu quisesse aplicar várias funções em um mesmo valor, seria possível? Bom, vamos criar outra função para ficar mais divertido, essa nova função será responsável por devolver metade do valor passado como argumento.

```
metade :: Int -> Maybe Int
metade x = Just (x `div` 2)
```

Agora respondendo a pergunta anterior, podemos sim aplicar várias funções ou uma mesma função repetidas vezes a um mesmo valor, basta utilizar o >>= de forma sequencial. Um exemplo usando as nossas duas funções seria:

```
ghci> Just 20 >>= soma10 >>= soma10 >>= metade >>= metade >>= metade
Just 5
```

Um questionamento interessante é, o que acontece caso continuamos a aplicar as funções, já que temos Just 5 e a função soma10 devolve Nothing para valores menores que 10? Bom, temos isso:

```
ghci> Just 20 >>= soma10 >>= soma10 >>= metade >>= metade >>= metade
>>= soma10 >>= soma10 >>= metade
Nothing
```

Como esperado, o resultado é Nothing, pois uma vez que temos Nothing, nenhuma função será capaz de mudar esse valor.

Agora que o >>= já está bem explicado, podemos ir para a última função da mônada, que é a >>. Para entender essa função melhor, podemos ver a implementação padrão dela:

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> n = m >>= \_ -> n
```

Podemos ver que ela nada mais é que a função bind em que a função já é pré definida, ela “ignora” o argumento e devolve um valor predeterminado. Por exemplo:

```
ghci> Just 10 >> Just 20
Just 20
ghci> Just 20 >> Nothing
Nothing
```

```
ghci> Nothing >> Just 20
Nothing
```

Nos dois primeiros casos, o argumento foi ignorado e foi devolvido um valor predefinido(Just 20 e Nothing, respectivamente). Já no terceiro caso, temos que lembrar a implementação do Monad Maybe mostrada anteriormente, caso Nothing seja o valor a, ao aplicar o bind a função não será aplicada e o retorno será Nothing, ou seja, a saída foi Nothing porque a função _ -> n não é aplicada.

do notation

As mônadas se tornaram tão importantes em Haskell que elas ganharam uma sintaxe própria chamada de *do notation*. A missão dessa sintaxe é juntar os valores monádicos em sequência, além de apresentar o código de maneira mais enxuta e de fácil leitura.
Para entender melhor sua verdadeira importância, vamos a uma expressão monádica simples:

```
ghci> Just 3 >>= (\x -> Just (x+10))
Just 13
```

Tranquilo, né? O x vira 3 dentro da função lambda e então é somado com 10. Agora, e se tivéssemos um >>= dentro da lambda?

```
ghci> Just 3 >>= (\x -> Just 10 >>= (\y -> Just (x+y)))
Just 13
```

Nesse caso, o x vira o 3 e o y vira 10, então eles são somados. Porém, já começa a ficar complicado de acompanhar e entender o que está acontecendo, uma alternativa seria escrever a expressão como uma função:

```
foo :: Maybe Int
foo = Just 3  >>= (\x ->
     Just 10 >>= (\y ->
     Just (x + y)))
```

De fato, fica um pouco mais fácil comparada com a anterior. No entanto, as funções lambdas continuam dificultando a leitura, e a do notation existe essencialmente para tirar elas do nosso código. Com a do notation, a função foo fica desse jeito:

```haskell
foo' :: Maybe Int
foo' = do
   x <- Just 3
   y <- Just 10
   Just (x + y)
```

Bem mais simples e de fácil entendimento. Com essa notação, nós podemos vincular valores monádicos a variáveis, usando o mesmo raciocínio que fizemos com >>= e funções lambdas só que de uma forma diferente. Perceba que usamos <- para atribuir o valor a variável nesse caso, assim como usamos o bind para atribuir em uma lambda, ademais, o último valor de uma *do expression* não se usa o <-, pois ele é o resultado da junção de todos os valores monádicos, portanto não é necessário fazer o bind. E como em uma expressão monódica, caso tenha um Nothing, o resultado também será Nothing:

```haskell
faztudo :: Maybe Int
faztudo = do
   x <- Just 20
   y <- soma10 x
   z <- soma10 y
   Nothing
   k <- metade z
   j <- metade k
   metade j
```

```
ghci> faztudo
Nothing
```

Leis

Para ter de fato uma mônada, não se pode apenas criar uma instância de mônada e é isso, existem três regras básicas que essas instâncias devem obedecer. Vamos dar uma olhadas nelas:

```haskell
return a >>= k                  =  k a
m        >>= return             =  m
m        >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

A primeira é bem simples, ela declara que pegando um valor a e usando >>= na função k, o resultado tem que ser o mesmo que pegar o valor a sem contexto e aplicar a função k nele.

```
ghci> Just 3 >>= (\x -> Just (x+2))
Just 5
ghci> (\x -> Just (x+2)) 3
Just 5
```

A segunda declara que ao usar o bind para a função de retorno, o valor tem que ser exatamente o mesmo valor original.

```
ghci> Just 3 >>= (\x -> return x)
Just 3
```

Já a terceira declara que não importa a ordem que as funções estão aninhadas, o resultado deve ser o mesmo.

```
ghci> Just 3 >>= (\x -> Just 5 >>= (\y -> Just (x + y)))
Just 8
ghci> Just 5 >>= (\x -> Just 3 >>= (\y -> Just (x + y)))
Just 8
```

Conclusão

Nesse pequeno tutorial sobre mônadas, discutimos sobre o conceito básico de mônadas em Haskell, as regras para que uma instância de mônada seja legítima, a sintaxe especial das mônadas, além de uma pequena implementação utilizando a mônada Maybe.

Referências

https://mvanier.livejournal.com/3917.html
https://learnyouahaskell.com/a-fistful-of-monads
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

Código usado: https://github.com/eduribeiro8/DGPT