Paradigmas de Programação

Prof. Maicon R. Zatelli

Haskell - Programação Funcional
Monads

Universidade Federal de Santa Catarina
Florianópolis - Brasil

Haskell

Princípio fundamental de linguagens funcionais: a avaliação de uma expressão deve sempre retornar o mesmo resultado, porém IO viola este princípio, isto é, causa efeitos colaterais. Ex: a leitura de dados da entrada poderá ler dados diferentes.

Monads (ou mônadas ou monoides) é um conceito vindo da *teoria das categorias* e é utilizado principalmente para garantir que este princípio seja mantido, além de outras coisas.

Computacionalmente, monads são estruturas de dados que encapsulam dados.

- Ex: **IO a**, onde um objeto do tipo **a** está encapsulado num objeto do tipo **IO a**, sendo que para acessar esse objeto do tipo **a** deve-se utilizar algumas operações pré-definidas, não sendo possível extraí-lo do tipo **IO a**.

Haskell

As linguagens funcionais utilizam os conceitos de monads para implementar estruturas procedimentais de forma funcional, assim permitindo alcançar:

- Sequencialidade (algumas coisas devem acontecer antes de outras)
- Manutenibilidade
- Modularidade

Haskell

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

Haskell

Leis para uma instância de Monad, ou seja, o que `return` e (>>=) (ligação/*bind*) devem fazer.

1. **return a**, return realiza uma ação vazia e simplesmente encapsula um objeto do tipo **a** em um monad.
2. **p >>= return = p** (return é um elemento neutro à direita para >>=), ou seja, return realiza uma ação vazia.
   - Se tenho um objeto monad p e combino com return, obtenho p (nada muda).
   - Isto é: m a >>= (a -> m a) = m a
3. return x >>= f = f x (return é também um elemento neutro à esquerda para >>=)
   - Se realizar return x >>= f, onde f é uma função, (e return x e o retorno de f são objetos monad), é o mesmo que aplicar a função f sobre x)
   - Isto é: m a >>= (a -> m b) = m b

Leis para uma instância de Monad, ou seja, o que `return` e (>>=) devem fazer.

4. p >>= \x -> q >>= \y -> r (>>= é associativa, ou seja, não importa qual >>= computamos primeiro)
   - Isto é: m a >>= a -> m b >>= b -> m c
   - Exemplo:
     (p >>= \x -> q) >>= \y -> r é igual a
     p >>= \x -> (q >>= \y -> r)

Onde Monads são úteis?

Exemplo: tratar operações que podem resultar em erro. Queremos avaliar expressões do tipo `div div 4 2 6`, a qual pode ser interpretada da forma `(div (div 4 2) 6)`

Formalmente, podemos obter a seguinte gramática para nossas expressões:

```
Expr ::- Number | Div Expr Expr
```

Problema: divisão por zero

Utilizando o Monad `Maybe`

Maybe como um tipo

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Read, Show)
```

Maybe como uma instância de Monad

```
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

Haskell

Inicialmente podemos pensar em uma função que recebe apenas dois números e retorna o resultado da operação `div` entre eles.

```
mydiv :: Int -> Int -> Maybe Int
mydiv n m | m == 0 = Nothing
          | otherwise = Just (n `div` m)
```

- Esta função retorna um `Maybe Int`, que pode ser `Nothing`, caso ocorra divisão por zero ou `Just x`, onde `x` é o resultado da divisão de `n` por `m`.

Para representar nossa expressão, podemos criar um novo tipo, o qual segue exatamente a gramática que definimos anteriormente.

```
data Expr = Val Int | Div Expr Expr
```

- Neste tipo, dizemos que uma `Expr` pode ser um `Val Int` ou `Div Expr Expr`, ou seja, uma expressão é um valor inteiro ou a divisão entre duas expressões.

Haskell

Podemos agora criar nossa primeira solução.

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case (eval x) of
                      Nothing -> Nothing
                      Just n -> case (eval y) of
                                     Nothing -> Nothing
                                     Just m -> (mydiv n m)
```

- Aqui criamos nossa função para avaliação, a qual recebe uma expressão e retorna talvez um inteiro.
- Caso a expressão recebida já é um inteiro, retorno Just n, onde n é o próprio valor inteiro.
- Caso contrário, avalio tanto a expressão a esquerda como a expressão a direita. Se alguma delas ocasionar divisão por zero, então retorno Nothing, caso contrário, retorno o resultado da divisão inteira de n por m.

Problema?

Haskell

Podemos agora criar nossa primeira solução.

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case (eval x) of
                      Nothing -> Nothing
                      Just n -> case (eval y) of
                                     Nothing -> Nothing
                                     Just m -> (mydiv n m)
```

- Aqui criamos nossa função para avaliação, a qual recebe uma expressão e retorna talvez um inteiro.
- Caso a expressão recebida já é um inteiro, retorno Just n, onde n é o próprio valor inteiro.
- Caso contrário, avalio tanto a expressão a esquerda como a expressão a direita. Se alguma delas ocasionar divisão por zero, então retorno Nothing, caso contrário, retorno o resultado da divisão inteira de n por m.

Problema? Código repetitivo e longo

Podemos pensar em usar conceitos de Monad, pois sabemos que o resultado da avaliação da parte esquerda `n` e da parte direita `m` são entradas para a função que computa a divisão inteira de `x` por `y`.

```
evalmonad :: Expr -> Maybe Int
evalmonad (Val n) = return n
evalmonad (Div x y) = evalmonad x >>=
                      (\n -> evalmonad y >>=
                          \m -> (mydiv n m))
```

- Note que o `return` simplesmente executa a ação vazia. Neste caso, seria o mesmo que escrever `Just n`.

Notação `do`

```
evalfinal :: Expr -> Maybe Int
evalfinal (Val n) = return n
evalfinal (Div x y) = do
    n <- evalfinal x
    m <- evalfinal y
    mydiv n m
```

- *Syntactic sugar* do Haskell para fazer o mesmo que fizemos anteriormente, porém com menos código.
- Compilador do Haskell fará a transformação para a sintaxe natural, respeitando as construções permitidas pela classe Monad.

Haskell

Agora, podemos chamar as funções criadas.

```
main = do
    print (mydiv 5 0)
    print (eval (Div (Val 5) (Val 0)))
    print (eval (Div (Val 5) (Val 2)))
    print (evalmonad (Div (Val 5) (Val 0)))
    print (evalmonad (Div (Val 5) (Val 2)))
    print (evalfinal (Div (Val 5) (Val 0)))
    print (evalfinal (Div (Val 5) (Val 2)))
```

Criando o próprio Monad

```
data MyMonad a = Val a
    deriving Show

instance Monad MyMonad where
    return = Val
    (>>=) (Val a) f = f a
```

- Aqui criamos um tipo `MyMonad`, que será uma instância da classe `Monad`. Precisamos então implementar as funções `return` e o operador `(»=)`.

Haskell

```
incrementadois :: Int -> MyMonad Int
incrementadois x =
    Val x >>=
        (\k -> incrementa k >>=
            (\z -> incrementa z >>=
                (\w -> return w)))

incrementa :: Int -> MyMonad Int
incrementa x = return (x + 1)
```

- A função `incrementa` foi criada para apenas incrementar 1 ao valor passado a ela como parâmetro e retornar um `MyMonad Int`, que neste caso será sempre `Val (x + 1)`.
- A função `incrementadois` foi criada para incrementar 2 ao valor passado a ela como parâmetro. Para isso, ela utiliza chamadas da função incrementa.

Notação `do`

```
incrementadoisDo :: Int -> MyMonad Int
incrementadoisDo x = do
    k <- Val x
    z <- incrementa k
    w <- incrementa z
    return w
```

- Aqui, a função `incrementadois` foi reescrita utilizando a notação do.

Haskell

Agora, podemos chamar as funções criadas.

```
main = do
    print (incrementadois 4)
    print (incrementadoisDo 4)
```

O Monad IO

IO () é um tipo para executar ações, ou seja, a avaliação de um objeto do tipo **IO ()** resulta na execução de uma ação. Assim, **IO ()** somente contém ações de saída. Note que o único elemento do tipo **IO ()** é **()**.

IO () é um tipo de dados abstrato e seus objetos encapsulados podem ser apenas acessados por meio de operações pre-definidas.

Exemplo de Função

```
putChar :: Char -> IO ()
```

O Monad IO

IO a é um tipo para executar ações, onde `IO a` é uma ação que encapsula (ou computa) um objeto do tipo **a**. Em outras palavras, **IO a** contém ações de "entrada", onde um objeto do tipo **a** é computado e encapsulado num objeto do tipo **IO a**.
IO a é um tipo de dados abstrato e seus objetos encapsulados podem ser apenas acessados por meio de operações pre-definidas.

Exemplo de Função

```
getChar :: IO Char
```

A avaliação da função `getChar` não é o caracter lido, mas a ação de ler um caracter, neste caso **return** pode ser definido como return :: a -> IO a, onde **return** simplesmente computa um objeto do tipo **a** e encapsula o mesmo num objeto do tipo **IO a**, e **a** é **Char**.

O Monad IO

Outros Exemplos de Funções

```
putStr :: Text -> IO ()
putStrLn :: String -> IO ()
getLine :: IO String
getContents :: IO String
print :: Show a => a -> IO ()
readFile :: String -> IO String
writeFile :: String -> String -> IO ()
```

Veja mais em:

`http://hackage.haskell.org/package/base-4.11.0.0/docs/System-IO.html`

```
ler :: Int -> IO String
ler 0 = return []
ler n = getChar >>=
            \x -> (ler (n -1)) >>=
                \xs -> return (x:xs)
```

- A função `ler` foi criada para ler n caracteres da entrada e retornar o valor lido.
- Caso deseja-se ler 0 caracteres, apenas retorna uma string vazia `[]`.
- Caso contrário, lê um caracter, armazena em `x`, lê os demais caracteres e armazena em `xs` (que conterá uma lista de caracteres lidos). Por fim, retorna a lista formada por `x:xs`.

Notação `do`

```
lerDo :: Int -> IO String
lerDo 0 = return []
lerDo n = do
    x <- getChar
    xs <- (lerDo (n -1))
    return (x:xs)
```

- Aqui, a função `lerDo` faz a mesma coisa que anteriormente, mas utiliza-se a notação **do**.

Haskell

Agora, podemos chamar as funções criadas.

```
main = do
    a <- ler 5
    print a
    b <- lerDo 5
    print b
```

Haskell - Alguns Links Úteis

- `https://www.haskell.org/tutorial/monads.html`
- `https://wiki.haskell.org/All_About_Monads`
- `https://wiki.haskell.org/Monad`
- `http://www.euclideanspace.com/maths/discrete/category/higher/monad/index.htm`

Analise os arquivos "*Monad* ***sem*** *Applicative e Functor*" e "*Monad* ***com*** *Applicative e Functor*" para verificar alterações na linguagem Haskell quanto ao uso de Monads.

- `https://wiki.haskell.org/Functor-Applicative-Monad_Proposal`

Haskell

Ver atividade no Moodle