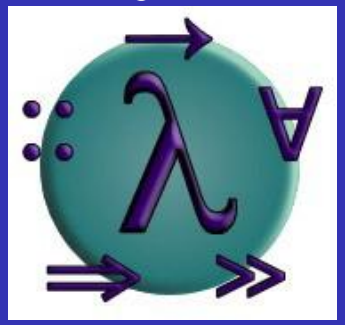
Programação Funcional



Capítulo 13

Mônadas

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2012.1

Tópicos

Mônadas

- Mônadas em Haskell podem ser pensadas como descrições de computação combináveis.
- Cada mônada representa um tipo diferente de computação.
- Uma mônada pode ser executada a fim de realizar a computação por ela representada e produzir um valor como resultado.

Operações monádicas básicas

- `(>>=)`
  Usado para realizar duas computações em sequência. Combina duas computações de forma que, quando a computação combinada é executada, a primeira computação é executada o seu resultado é passado para a segunda computação, que então é executada usando (ou dependendo de) aquele valor.
- `return`
  `return x` é uma computação que, quando executada, apenas produz o resultado `x`.

A classe Monad

- Uma mônada pode ser declarada como uma classe em Haskell:

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

  p >> q = p >>= \_ -> q

  fail msg = error msg
```

- Nesta classe de tipo a variável restrita `m` representa um construtor de tipo (de aridade um), e não um tipo!
- Um construtor de tipo é uma mônada se existirem as operações `return` e `(>>=)` que permitem combinar valores do tipo em sequência.

Leis das mônadas

Além de implementar as funções da classe **Monad**, todas as mônadas devem obedecer as seguintes leis, dadas pelas equações:

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

Tópicos

Entrada e saída

- Uma ação de entrada e saída é representada pelo tipo `IO a`.
- `IO a` é um tipo abstrato em que pelo menos as seguintes operações estão disponíveis:
  - Operações primitivas, como por exemplo:

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

  - Operações para combinar ações de entrada e saída:

```
return :: a -> IO a
```

    - `return x` é uma ação de E/S que quando executada não interage com o mundo e retorna `x`.

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

    - `p >>= q` é uma ação de E/S que quando executada, executa primeira a ação `p` e em seguida aplica a função `q` no resultado de `p`.
    - O corpo da função `q` é a segunda ação da sequência.
    - O resultado da primeira ação pode ser usado na segunda ação porque ele é passado como argumento para a função `q`.

Entrada e saída (cont.)

- As funções `return` e `(>>=)` caracterizam a classe de tipos chamada **mônada**.
- As demais funções primitivas são específicas de ações de entrada e saída.
- Assim `IO` é uma mônada e `x :: IO a` é uma ação monádica.

Tópicos

Notação do

- Tipicamente computações são construídas a partir de longos encadeamentos dos operadores `(>>)` e `(>>=)`.
- Haskell oferece a **expressão** `do`, que *permite combinar várias computações a serem executadas em sequência* usando uma notação mais conveniente.
- Uma expressão `do` é uma extensão sintática do Haskell e sempre pode ser reescrita como uma expressão mais básica usando os operadores de sequenciamento `(>>)` e `(>>=)` e a expressão `let`.

Exemplo de notação do

Um programa para ler dois números e exibir a sua soma:

```
module Main (main) where

import System.IO ( stdout, hSetBuffering, BufferMode(NoBuffering) )

main :: IO ()
main = do { hSetBuffering stdout NoBuffering;
            putStr "Digite um número: ";
            s1 <- getLine;
            putStr "Digite outro número: ";
            s2 <- getLine;
            putStr "Soma: ";
            putStrLn (show (read s1 + read s2))
          }
```

Regra de layout com a notação do

- A expressão **`do`** pode usar ***layout*** em sua estrutura sintática, de maneira semelhate à expressão **`let`** e às cláusulas **`where`**.
- As regras de *layout*, por meior do uso de indentação adequada, permitem omitir as chaves `{` e `}` usadas para delimitar o corpo da expressão `do` e os pontos-e-vírgula `;` usados para separar as ações que compõem o seu corpo.
- Neste caso todas as ações devem começar na mesma coluna, e se continuarem nas linhas seguintes, não podem usar colunas menores que esta coluna.

Exemplo de notação do usando *layout*

Um programa para ler dois números e exibir a sua soma:

```
module Main (main) where

import System.IO ( stdout,
                   hSetBuffering, BufferMode(NoBuffering) )

main :: IO ()
main = do hSetBuffering stdout NoBuffering
          putStr "Digite um número: "
          s1 <- getLine
          putStr "Digite outro número: "
          s2 <- getLine
          putStr "Soma: "
          putStrLn (show (read s1 + read s2))
```

Tradução da expressão do

Código escrito usando a notação do é transformado automaticamente pelo compilador em expressões ordinárias que usam as funções `(>>=)` e `(>>)` da classe `Monad`, e a expressão `let`.

Tradução da expressão do (cont.)

Quando houver uma única ação no corpo:

```
do { ação }
≡
ação
```

Exemplo:

```
do putStrLn "Bom dia, galera!"
≡
putStrLn "Bom dia, galera!"
```

Observe que neste caso não é permitido usar a forma

```
padrão <- ação
```

pois não há outras ações que poderiam usar variáveis instanciadas pelo casamento de padrão.

Tradução da expressão do (cont.)

Quando houver duas ou mais ações sem casamento de padrão na primeira ação:

```
do { ação ; resto }
≡
ação >> do { resto }
```

Exemplo:

```
do putStrLn "um" ; putStrLn "dois"
≡
putStrLn "um" >> do putStrLn "dois"
≡
putStrLn "um" >> putStrLn "dois"
```

Tradução da expressão do (cont.)

Quando houver duas ou mais ações com casamento de padrão na primeira ação:

```
do { padrão <- ação ; resto }
≡
ação >>= \padrão -> do { resto }
```

Exemplo:

```
do x <- getLine ; putStrLn ("Você digitou: " ++ x)
≡
getLine >>= \x -> do putStrLn ("Você digitou: " ++ x)
≡
getLine >>= \x -> putStrLn ("Você digitou: " ++ x)
```

Tradução da expressão do (cont.)

Quando houver duas ou mais ações com declaração local na primeira ação:

```
do { let declarações ; resto }
≡
let declarações in do { resto }
```

Exemplo:

```
do let f xs = xs ++ xs ; putStrLn (f "abc")
≡
let f xs = xs ++ xs in do putStrLn (f "abc")
≡
let f xs = xs ++ xs in putStrLn (f "abc")
```

Tópicos

Computações que podem falhar

- A mônada **Maybe** representa computações que podem falhar.
- Declaração:

```haskell
data Maybe a = Nothing | Just a
```

```haskell
instance Monad Maybe where
  return = Just

  Nothing >>= f = Nothing
  Just x  >>= f = f x
```

Computações que podem falhar (cont.)

- Exemplo: agenda telefônica

```
agenda :: [(String,String)]
agenda = [ ("Bob",   "01788 665242"),
           ("Fred",  "01624 556442"),
           ("Alice", "01889 985333"),
           ("Jane",  "01732 187565") ]
```

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup x []                        = Nothing
lookup x ((y,v):ys) | x == y       = Just v
                    | otherwise = lookup x ys
```

- Queremos procurar dois itens na agenda:
  - se algum dos itens não for encontrado, a operação falha
  - se ambos os itens forem encontrados, resulta no par formado pelos valores correspondentes

```
lookup2 :: String -> String -> Maybe (String,String)
```

Computações que podem falhar (cont.)

- Inspecionando diretamente a estrutura de dados:

```
lookup2 a b = case lookup a agenda of
                Nothing -> Nothing
                Just x  -> case lookup b agenda of
                             Nothing -> Nothing
                             Just y  -> Just (x,y)
```

- Usando operações monádicas sem a notação do:

```
lookup2 a b = lookup a agenda >>= \x ->
              lookup b agenda >>= \y ->
              return (x,y)
```

- Usando operações monádicas com a notação do:

```
lookup2 a b = do x <- lookup a agenda
                 y <- lookup b agenda
                 return (x,y)
```

Tópicos

Expressões aritméticas

- Considere o tipo `Exp` que representa uma expressão aritmética:

```
data Exp = Cte Integer
         | Som Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
         deriving (Read, Show)
```

- Uma expressão aritmética é:
  - uma constante inteira, ou
  - a soma de duas expressões, ou
  - a diferença de duas expressões, ou
  - o produto de duas expressões, ou
  - o quociente de duas expressões.

Exemplos de expressões aritméticas

```
-- 73/(3+3) * 8
expOk = Mul (Div (Cte 73)
                 (Som (Cte 3) (Cte 3)))
             (Cte 8)

-- 73/(3-3) * 8
expProblema = Mul (Div (Cte 73)
                       (Sub (Cte 3) (Cte 3)))
                   (Cte 8)
```

Avaliação de expressões aritméticas

- Um avaliador simples de expressões aritméticas:

```
avalia :: Exp -> Integer

avalia (Cte x) = x
avalia (Som a b) = avalia a + avalia b
avalia (Sub a b) = avalia a - avalia b
avalia (Mul a b) = avalia a * avalia b
avalia (Div a b) = div (avalia a) (avalia b)
```

Avaliação de expressões aritméticas (cont.)

- Avaliando as expressões anteriores:

```
*Main> avalia expOk
96
*Main> avalia expProblema
*** Exception: divide by zero
```

- A segunda expressão não pode ser avaliada, pois a divisão por zero leva a um resultado indefinido.

Exercícios

Exercício 1

Redefina a função `avalia` para que ela não produza uma exceção quando em sua estrutura houver divisão por zero.
A função deve retornar uma indicação de falha ou sucesso juntamente com o seu valor em caso de sucesso.
Inspecione os resultados das avaliações das subexpressões diretamente usando análise de casos.

Exercício 2

Modifique a função `avalia` do exercício anterior para usar operações monádicas ao invés de inspecionar os resultados das avaliações das subexpressões diretamente usando análise de casos.
Não use a notação do.

Exercício 3

Modifique a função `avalia` do exercício anterior para usar a notação do para realizar as operações monádicas.

Fim