Mônadas, Entrada e Saída

Programação Funcional

Prof. Maycon Amaro

Nas aulas anteriores. . .

- Vimos construtores de tipos (lista, opcional, tupla)

```
x :: Maybe Int
x = Just 5
```

- Vimos como aplicar funções a valores que estão *contextualizados* por um construtor de tipos, por meio de funtores.

```
ghci> fmap (>0) (Just 5)
Just True
```

- E também como realizar a aplicação quando as funções também estão *contextualizadas*, por meio de funtores aplicativos.

```
ghci> (Just even) <*> (Just 5)
Just True
```

Funções podem inserir estrutura

Considere a função abaixo, que calcula o predecessor de um número positivo.

```
predPos :: Int -> Maybe Int
predPos x
    | x > 0      = Just (x - 1)
    | otherwise = Nothing
```

Essa função inevitavelmente *contextualiza* o retorno, o colocando na estrutura do `Maybe`.

```
ghci> predPos 5
Just 4
ghci> predPos 2
Just 1
ghci> predPos 0
Nothing
ghci> predPos (-1)
Nothing
```

Se aplicarmos essa função a um valor contextualizado, acabaremos com estruturas aninhadas

```
ghci> fmap predPos (Just 5)
Just (Just 4)
ghci> fmap predPos (Just 0)
Just Nothing
ghci> (Just predPos) <*> (Just 5)
Just (Just 4)
ghci> (Just predPos) <*> (Just 0)
Just Nothing
```

Também é difícil encadear a função

```
ghci> let x = Just 5
ghci> let y = fmap predPos x
ghci> let z = (fmap . fmap) predPos y
ghci> let w = (fmap . fmap . fmap) predPos z
ghci> w
Just (Just (Just (Just 2)))
```

Como lidar com isso?

Mônadas

```
type Monad :: (* -> *) -> Constraint
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  {-# MINIMAL (>>=) #-}
```

Note que é necessário ser instância de `Applicative` para ser instância de `Monad`.

return

- Este termo é bastante comum em outras linguagens, mas não necessariamente quer dizer a mesma coisa aqui
- A função `return` se comporta exatamente como `pure` para `Applicative`

```
ghci> pure 5 :: Maybe Int
Just 5
ghci> return 5 :: Maybe Int
Just 5
ghci> pure 5 :: [Int]
[5]
ghci> return 5 :: [Int]
[5]
ghci> pure 5 :: (String, Int)
("",5)
ghci> return 5 :: (String, Int)
("",5)
```

bind

- É o nome do nosso operador de interesse >>=
- Lida com funções que inserem estrutura, sem causar os aninhamentos
- Intuitivamente, poderíamos dizer que ele *extrai* os valores antes de realizar a aplicação

```
ghci> (Just 5) >>= predPos
Just 4
ghci> (Just 2) >>= predPos
Just 1
ghci> (Just 0) >>= predPos
Nothing
```

- Mas podemos trabalhar com a ideia de que ele está *combinando* as estruturas aninhadas.

```
join :: Monad m => m (m a) -> m a
```

Utilizando `join` e `fmap`, podemos recriar o comportamento de `>>=`.

```
ghci> join (Just (Just 5))
Just 5
ghci> join [[1, 2], [3, 4]]
[1,2,3,4]
ghci> join ("Amar", ("Elo", 5))
("AmarElo",5)
ghci> join (fmap predPos (Just 5))
Just 4
```

Problema resolvido?

Agora é bastante simples encadear funções que inserem estrutura

```
ghci> Just 5 >>= predPos >>= predPos >>= predPos
Just 2
ghci> Just 1 >>= predPos >>= predPos >>= predPos
Nothing
ghci> return 5 >>= predPos >>= predPos >>= predPos
Just 2
```

Mais exemplos de encadeamentos

```
-- Lista
ghci> predPos x = [x, x-1]
ghci> return 5 >>= predPos >>= predPos >>= predPos
[5,4,4,3,4,3,3,2]
-- Tupla
ghci> predPos x = ("-", x-1)
ghci> return 5 >>= predPos >>= predPos >>= predPos
("---",2)
--- Either (Variant)
ghci> predPos x = Right (x-1)
ghci> return 5 >>= predPos >>= predPos >>= predPos
Right 2
```

do Notation

Um *syntax sugar* para facilitar a escrita do encadeamento de >>=

```
do
  x <- predPos 5
  y <- predPos x
  return y
```

É o mesmo que

```
return 5 >>= predPos x >>= predPos y
```

Aplicações de Mônadas

- `State` é uma mônada capaz de representar estados e mudanças de estado de uma maneira segura e funcional.
- *Parser combinators*, através do uso de `State` e do conhecimento sobre mônadas, permite criar funções de parsing compactas e poderosas.
- Há ainda outras classes de tipos de interesse, que fazem uso de funtores aplicativos e mônadas, como `Reader`, `Traversable`, `Foldable`.
- Vamos ver um importante uso de mônadas: `IO`.

Entrada e Saída

Introdução

- A programação funcional tenta abolir efeitos colaterais ao máximo.
- Imprimir algo na tela, escrever num arquivo, receber entrada do teclado são exemplos de **efeitos**.
- Mas para criar programas que interagem com o usuário, precisamos deles.

A mônada IO

- Encapsula valores originados de operações de entrada e saída
- A função `putStrLn` exibe uma string na tela

```
ghci> putStrLn :: String -> IO ()
```

- Seu retorno é um valor fixo (o tipo unitário!) encapsulado por `IO`
- Por causa do tipo, o compilador rejeitaria qualquer função pura que tentasse imprimir algo na tela

Um compilador de C/C++ permitiria a função abaixo

```c
int soma2(int x) {
  printf("Somei 2");
  return x + 2;
}
```

Mas um compilador de Haskell não

```haskell
-- Será rejeitado pelo compilador
soma2 :: Int -> Int
soma2 x =
  do
    putStrLn "Somei 2" -- retorna IO ()
    return (x + 2) -- retorna IO Int
```

Se quisermos este comportamento, temos que mudar o tipo da função

```
soma2 :: Int -> IO Int
soma2 x =
  do
    putStrLn "Somei 2"
    return (x + 2)
```

O tipo da função nos entrega que não podemos confiar que ela é pura

Se estivéssemos à procura da causa de um *bug* que está modificando arquivos no sistema, quais dessas funções poderiam ser a causa?

```haskell
f :: Int -> String -> Int
-- ...

g :: Bool -> (Bool, Int) -> String
-- ...

h :: String -> IO String
-- ...

j :: Monad m => m a -> (a -> m b) -> m b
-- ...
```

Boa sorte procurando a causa...

```
int f(char*, int) { ... }

bool g(bool, int) { ... }

char* h(char*) { ... }

double j (int[], int) { ... }
```

A função `main`

- É do tipo `IO ()`, ou seja, deve encerrar com alguma operação de entrada e saída que não produz valor útil.

```
main :: IO ()
main = putStrLn "Olá, mundo!" -- retorna IO ()
```

```haskell
-- será reijeitado
main :: IO ()
main =
  do
    putStrLn "Olá, mundo!"
    return 2 -- retorna IO Int

-- será aceito
main :: IO ()
main =
  do
    putStrLn "Olá, mundo!" -- retorna IO ()
    return () -- retorna IO ()
```

Mais funções IO

```haskell
-- Lê uma linha do teclado
getLine :: IO String

-- Imprime instâncias de Show
print :: Show a => a -> IO ()

-- Lê um símbolo do teclado
getChar :: IO Char

-- Lê o conteúdo de um arquivo
readFile :: FilePath -> IO String

-- Cria/apaga um arquivo e escreve nele
writeFile :: FilePath -> String -> IO ()
```

Exemplo

```haskell
main :: IO ()
main = getLine >>= print . length
```

Usando do-notation

```haskell
main :: IO ()
main
  = do
    s <- getLine
    print (length s)
```

Lendo números do teclado

`read` converte `String` em outros tipos que sejam instância de `Read`

```
main :: IO ()
main =
  do
    putStr "Digite um número: "
    x <- getLine
    putStr "O sucessor é "
    print ((read x) + 1)
```

Se o usuário digitar algo não numérico neste caso, o programa se encerra.

Com `readIO` um eventual erro de conversão é passado para a mônada `IO` ao invés de encerrar o programa.

```
main :: IO ()
main =
  do
    putStr "Digite um número: "
    x <- getLine
    putStr "O sucessor é "
    y <- readIO x
    print (y + 1)
```

A função `readLn` combina `getLine` e `readIO`

```
main :: IO ()
main =
  do
    putStr "Digite um número: "
    x <- readLn
    putStr "O sucessor é "
    print (x + 1)
```

Sem do-notation

```
main :: IO ()
main = putStr "Digite um número: "
       >> readLn
       >>= \x -> putStr "O sucessor é "
       >> print (x + 1)
```

Conclusão

- `IO` é um construtor de tipos e instância de `Functor`, `Applicative` e `Monad`
- É possível utilizar `fmap` e `<*>` sobre ele também
- Há várias bibliotecas para auxiliar na interação com usuário, como a `Text.Printf`
- Um bom programa será escrito com várias funções puras, e apenas o estritamente necessário será feito em funções contextualizadas com `IO`
- `IO` também é capaz de propagar um erro de entrada e saída, para que seja tratado apenas quando necessário

Recomendação

Leiam o post *Funtores, Aplicatives e Mônadas explicados com desenhos* cujo link está no Moodle.

Dúvidas?