Programação Funcional e Lógica

Mônadas

Prof. Ricardo Couto A. da Rocha
rcarocha@ufg.br
UFG – Regional de Catalão

- *Baseado no curso "Introduction to Haskell" de Brent Yorgey (University of Pennsylvania) E*
- *"Learn Haskell for Great Good". Miran Lipovaca. http://learnyouahaskell.com/*
- *https://en.wikibooks.org/wiki/Haskell/Understanding_monads*

Leitura de Referência

Aprender Haskell será um grande bem para você - Livro-tutorial online (tradução do livro *"Learn You a Haskell for Great Good!"* de Miran Lipovaca)

- **Capítulo 12: Um punhado de Monads**

Understanding Monads

https://en.wikibooks.org/wiki/Haskell/Understanding_monads

Roteiro

- Conceito e Propósito Geral
- Definição
- Motivação: Mônadas **`Maybe`**
- Classe **`Monad`**
- Açucar sintático **`do`**
- Exemplos de Mônadas: Listas
- Leis sobre Mônadas (Monádicas)

Mônada

- Maneira de **estruturar computações** - em linguagens funcionais - levando em conta
  - **Valores** usados para computações
  - **Sequências** de computações usando os valores
- Estabelece um **bloco de construção de programas** que descreve **sequências de computações**
  - Lembrando a estrutura de um programa imperativo.
- Mônada permite determinar **com uma computação combinada forma outra computação**, simplificando o trabalho do programador em descrevê-lo manualmente sempre que necessário.

Mônada

- Estratégia de combinar computações em computações mais complexas.
- Papel central no sistema de I/O de Haskell
- Entender o funcionamento permite tornar o código melhor e melhorar suas funções.
- Três propriedades úteis para estruturar programs funcionais:
  - **Modularidade**: Computações formadas de computações mais simples
  - **Flexibilidade**: programas são mais adaptáveis do que aqueles escritos sem mônadas
  - **Isolamento**: estruturas de estilo imperativo de programação ficam isoladas do corpo do programa funcional.
    - Útil para incorporar efeito colateral e estado em linguagens puramente funcionais.

Definição

- Mônada é definido por:
  - Construtor de tipo `m` (no caso de IO é `IO`)
  - Uma função `return`
  - Um operador `(>>=)`, chamado de "bind" (ou "associe/a")
- A classe de tipos Monad especifica um mônada e suas funções são assim declaradas:

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

- Um mônada também deve obedecer algumas propriedades.

Motivação: Mônada Maybe

- Para motivar o uso de mônadas, considere **Maybe**
  - Mostraremos um exemplo de uso
  - Como ele pode ser simplificado, pelo fato de que **Maybe** é um mônadas.
- Está disponível no módulo **Data.Maybe** para representar um dado que pode ou não existir.

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

- **Exemplo**: a função **cabecaLista** abaixo nem sempre é capaz de retornar um valor. Por exemplo, se a lista for vazia.

```
cabecaLista :: [a] -> a
cabecaLista (a:as) = a
cabecaLista [] = ????
```

Motivação: Mônada Maybe

- Usando **Maybe**

```
cabecaLista :: [a] -> Maybe a
cabecaLista (a:as) = Just a
cabecaLista [] = Nothing
```

- Mais detalhes:
  https://wiki.haskell.org/Maybe
- Ela é particularmente interessante para lidar com IO (entrada e saída) que é intrinsecamente sujeita a falhas e erros.

Exemplo de uso Maybe

- Considere duas funções de um banco de dados genealógico:

```
pai :: Pessoa -> Maybe Pessoa
mae :: Pessoa -> Maybe Pessoa
```

- Caso um nome não seja encontrado (por exemplo, o banco está incompleto), então a função retorna Nothing.
- Combinação das funções para consultas de avos → Avô materno

```
avôMaterno :: Pessoa -> Maybe Pessoa
avôMaterno p =
    case mae p of
        Nothing -> Nothing
        Just m -> pai m
```

Fonte: https://en.wikibooks.org/wiki/Haskell/Understanding_monads

Exemplo de uso Maybe

- Função que procura se ambos os avôs estão no banco de dados

```
ambosAvôs :: Pessoa -> Maybe (Pessoa, Pessoa)
ambosAvôs p =
    case pai p of
        Nothing -> Nothing
        Just paiP ->
            case pai paiP of
               Nothing -> Nothing
               Just avo1 ->                    -- encontrou 1o. avô
                   case mae p of
                       Nothing -> Nothing
                       Just maeP ->
                           case pai maeP of
                               Nothing -> Nothing
                               Just avo2 -> -- encontrou 2o.avô
                                   Just (avo1, avo2)
```

Código extenso, repetitivo e deselegante para realizar muito pouco!

Maybe como Mônada

- **Maybe** também é uma instância de mônadas e segue a seguinte especificação

```
instance Monad Maybe
 (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
 (>>) :: Maybe a -> Maybe b -> Maybe b
 return :: a -> Maybe a
```

```
return x  = Just x

(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
            Nothing -> Nothing
            Just x  -> g x
```

Maybe como Mônada

- **Especificação de Maybe**

```
instance Monad Maybe

   …
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
             Nothing -> Nothing
             Just x  -> g x
```

Parte do código pode ser reescrita da seguinte maneira:

```
avôMaterno p = mae p >>= pai
```

Maybe como Mônada

- A função de retorno dos dois avôs maternos pode ser reescrita da seguinte maneira explorando operações sobre o mônada **Maybe**

```
ambosAvôs p =
  pai p >>=
    (\paiP -> pai paiP >>=
      (\avo1 -> mae p >>=
        (\maeP -> pai maeP >>=
           (\avo2 -> return (avo1,avo2) ))))
```

- O código ficará ainda mais legível/simples ao usarmos um açucar sintático

Classe de Tipos Monad

- Definição da classe **Monad**

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

   (>>) :: m a -> m b -> m b
   fail :: String -> m a
```

- Operador **(>>)** é chamado "então" ("then") e tem a seguinte implementação

```
    m >> n = m >>= \_ -> n
```

Classe de Tipos Monad

- Operador (>>) é chamado "então" ("then") e tem a seguinte implementação

```
m >> n = m >>= \_ -> n
```

- Essencialmente, ele é usando quando uma segunda computação "n", executada depois de "m", independe do resultado da primeira.
- Simplificação para não precisarmos repetir o código do operador, quando esse caso se aplicar.
- Exemplo:

```
escrevaDuasVezes :: String -> IO ()
escrevaDuasVezes str = putStrLn str >> putStrLn str
```

Açucar sintático do

- A sintaxe de uso dos operadores `(>>)` e `(>>=)` não é muito agradável para escrever nossos programas Haskel
- A linguagem nos oferece um açúcar sintático, usando a construção **do.**

```
putStrLn "um" >> putStrLn "dois"
```

tem o mesmo significado de

```
do
   putStrLn "um"
   putStrLn "dois"
```

- Quando o compilador encontra esse trecho de código ele o transforma o trecho de código anterior, usando o operador (>>).

Açucar sintático do

- Açucar sintático para o operador (>>=)

```
do
  padrão <- ação
```

corresponde ao código com o operador (>>=)

```
acao >>= \padrao -> código seguinte …
```

- Exemplo

```
do
   x <- getLine
   putStrLn ("Você digitou: " ++ x)
```

tem o mesmo significado de

```
getLine >>= \x -> putStrLn ("Você digitou: " ++ x)
```

Reescrevendo o uso MayBe

- No caso da função de retorno dos dois avôs maternos que foi escrita

```
ambosAvôs p =
    pai p >>=
        (\paiP -> pai paiP >>=
            (\avo1 -> mae p >>=
                (\maeP -> pai maeP >>=
                    (\avo2 -> return (avo1,avo2) ))))
```

pode ser escrita usando do da seguinte maneira

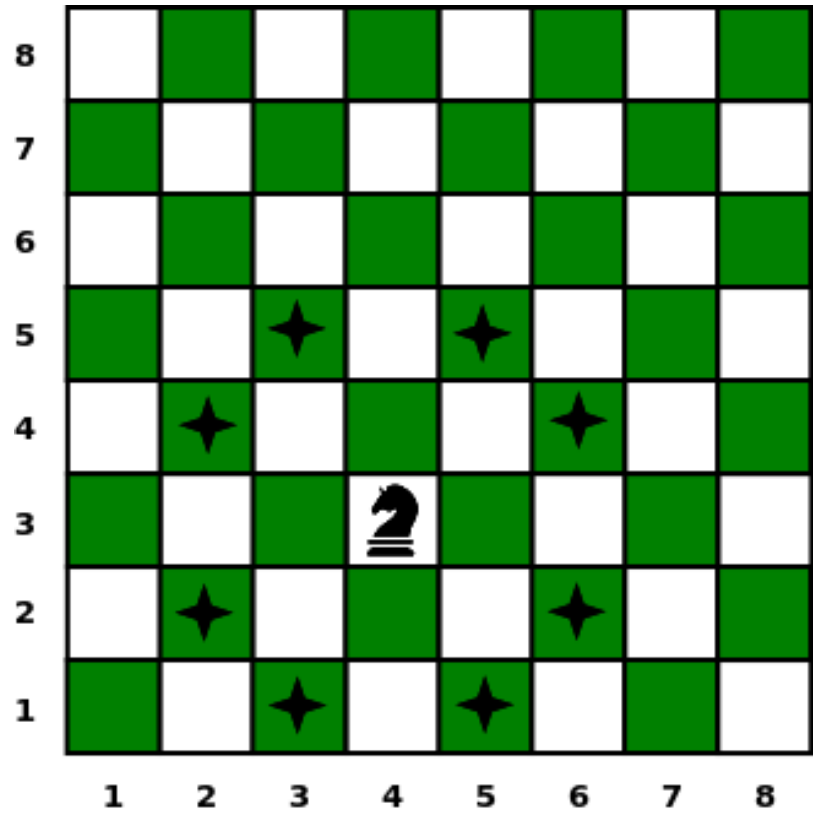
```
ambosAvôs p = do
    paiP <- pai p
    avo1 <- pai paiP
    maeP <- mae p
    avo2 <- pai maeP
    return (avo1, avo2)
```

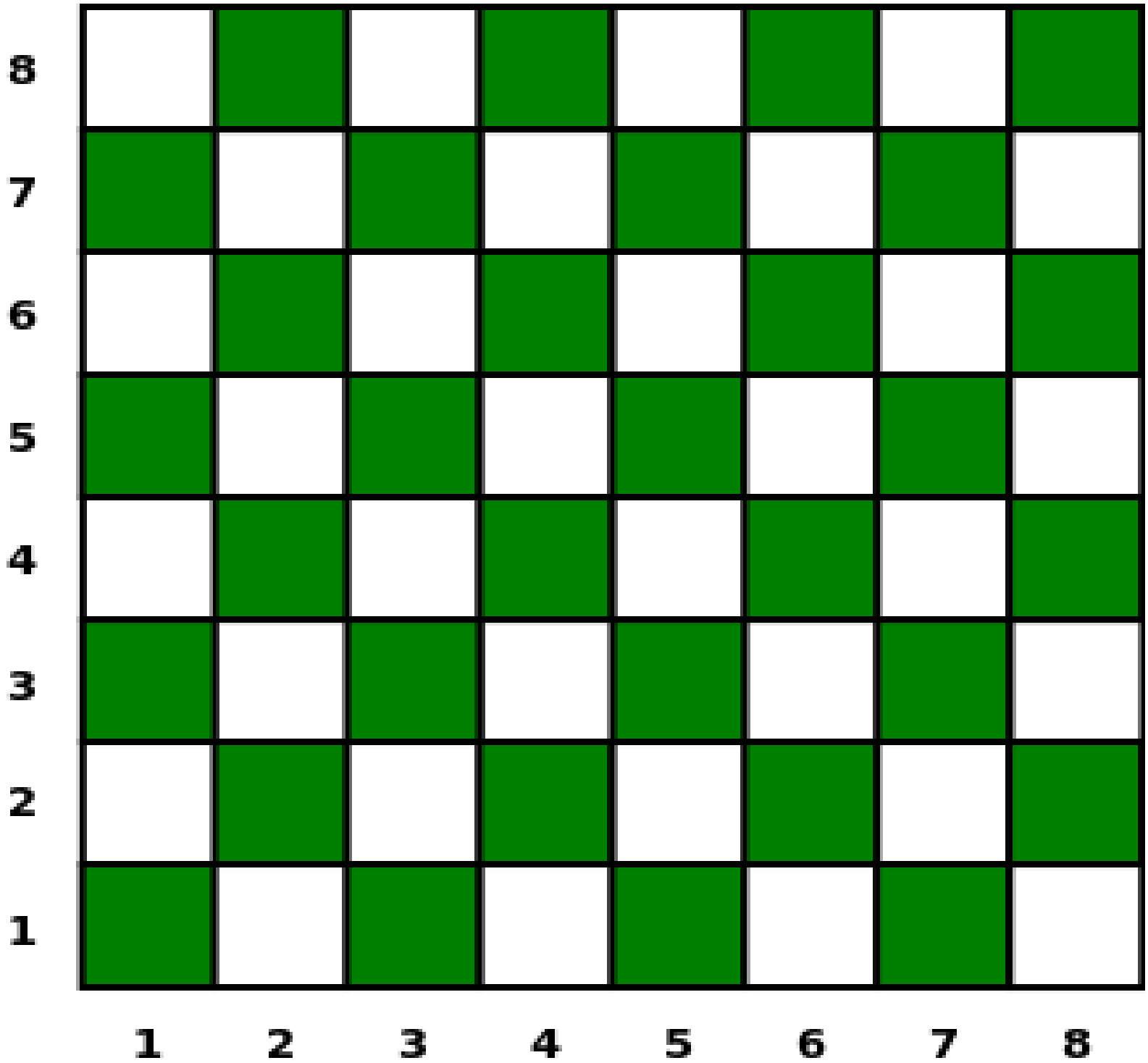
Exemplos de Mônadas

Mônada (Monad)	Semântica Imperativa
`Maybe`	Exceções (anônimas)
`Error`	Exceções (erros com descrição)
`IO`	Entrada e Saída
`[]` (listas)	Não-determinismo
`Reader`	Ambiente
`Writer`	Logger
`State`	Estado global

Exemplo de Mônadas: Listas

- Problema: *"Missão do Cavalo"*
  - Descobrir se uma certa posição do tabuleiro de xadrez pode ser alcançada em até n jogadas de um cavalo, partindo de um ponto inicial




8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8

Listas como Mônadas

- Listas em Haskell também são instâncias de Mônadas (**Monad**)
- A idéia é que listas são a mecanismo usual para modelar computações não-determinísticas
  - Que podem resultar um número arbitrário de resultados de número não conhecido a priori
- Modelagem de computações
  - **Maybe** → uma computação que pode ou não ocorrer (pode falhar), ou com zero ou um resultado
  - Listas → computações que podem ter zero, um ou um número arbitrário de resultados

Operações Monádicas em Listas

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

   (>>) :: m a -> m b -> m b
   fail :: String -> m a
```

Bind

Then (Então)

- Lista `[a]` é mônada `m a` na medida em que pode ser `[] a` , onde `[]` é o construtor de tipo.
- Operações

```
return x = [x]
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = concat (map f xs)
```

Operações Monádicas em Listas

```
return x = [x]
```

- Cria mônada a partir de dado não-mônada

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = concat (map f xs)
```

- Bind deve gerar sempre um mônada lista
- **concat** junta o conteúdo de listas em lista única
- **f** é o gerador de resultados de computações e entradas são valores na lista
- Resultado da aplicação é concatenado em única lista

Bind de Listas: Exemplo

- Usar `replicate 3` como `f`
- Aplicar bind em listas sucessivas

```
let reproducao = replicate 3
["coelho"] >>= reproducao
["coelho"] >>= reproducao >>= reproducao
["coelho"] >>= reproducao >>= reproducao >>=
reproducao
```

Bind de Listas: Exemplo

- Reescrever o código com o açucar sintático

  do

```
["coelho"] >>= reproducao >>= reproducao >>=
reproducao
```

- Usando **do**

```
do
   adao <- ["coelho"]
   primeiraGer <- reproducao inicio
   segundaGer <- reproducao primeiraGer
   terceiraGer <- reproducao segundaGer
```

Operador then (então)

```
m >> n = m >>= \_ -> n
```

- que para listas equivale a

```
m >> n = concat (map (\_ -> n) m)
```

- Quais são os resultados das avaliações abaixo?

```
[1,2,3] >> [8,9]
[1,2,3] >> "abcde"
[1,2,3] >> ["abcde"]
[1,2,3] >> "abcde" >> [1,0]
```

- Como a último código é descrito com **do**?

Exemplo

- **Lab 1.9**: Resolver o problema "Missão do Cavalo"
  - Criar uma função `proxCavalo (x,y)` que retorna uma lista com as possíveis posições de um cavalo a partir de `(x,y)`
    - Se retorna lista, retorna mônada
  - Função deve ser compatível com a função de computação do operador `(>>=)` de lista
  - Criar código que retorna as possíveis posições de um cavalo, após 3 movimentos, usando mônadas

Leis sobre Mônadas

- Qualquer tipo que implemente as operações pode, pela linguagem, ser uma instância de Monad.
- Isso não faz, entretanto, o tipo um mônada do ponto de vista matemático → o tipo precisa obedecer a três leis
- Essa verificação deve ser feita pelo programador
- Leis sobre mônadas
  - **Primeira Lei**: Identidade à esquerda

    `return x >>= f` é a mesma coisa que `f x`

Leis sobre Mônadas

- Leis sobre mônadas
  - **Segunda Lei**: Identidade à direita

    `m >>= return` é o mesmo que `m`
  - **Terceira Lei**: Associatividade

    `(m >>= f) >>= g`

    é equivalente a

    `m >>= (\x -> f x >>= g)`