Programação Funcional e Lógica

Funções de Alta Ordem em Haskell

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*
- *"Introduction to Functional Programming using Haskell". Richard Bird.*
- *"Learn Haskell for Great Good". Miran Lipovaca. http://learnyouahaskell.com/*

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 6: Funções de Alta Ordem**

`http://haskell.tailorfontela.com.br/higher-order-functions`

Funções de Alta Ordem

- Em Haskell, funções são cidadãos de primeira classe
  - Podemos manipular funções tal como manipulamos inteiros, booleanos.
  - Diversos operadores são oferecidos para aplicações especializadas de funções
    - (.) → função composição
    - ($) → função aplicação
- Falamos sobre `map`, `filter`, operador `(.)`

zipWith

- Considere a função **zipWith**

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

- Qual é o resultado da aplicação nos seguintes casos?

```
zipWith' (+) [4,2,5,6] [2,6,2,3]
zipWith' max [6,3,2,1] [7,3,1,5]
zipWith' (++) ["foo ", "bar ", "baz "]
              ["fighters", "hoppers", "aldrin"]
zipWith' (*) (replicate 5 2) [1..]
zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]]
                        [[3,2,2],[3,4,5],[5,4,3]]
```

Funções Anônimas

- Suponhamos que precisamos implementar a seguinte função para manter inteiros maiores que 100 em uma lista

```
greaterThan100 :: [Integer] -> [Integer]
```

- Exemplo de uso

```
greaterThan100 [1,9,349,6,907,98,105] = [349,907,105]
```

- Sugestão de implementação

```
gt100 :: Integer -> Bool
gt100 x = x > 100

greaterThan100 :: [Integer] -> [Integer]
greaterThan100 xs = filter gt100 xs
```

- A função **gt100** foi criada uma única vez e, provavelmente, nunca mais será usada. Seu único propósito foi auxiliar a função **greaterThan100**.

Funções anônimas

- Ao invés de criar uma função nova nomeada, podemos usar uma **função anônima** ou **abstração lambda** em Haskell

$$\lambda x \rightarrow x^2$$

- Expressão matemática que cria uma função que retorna para um parâmetro **x** o seu quadrado x^2.
- A função gerada não tem nome. É como um valor literal de uma função.
- Em Haskell, essa função é descrita usando \ como substituto da letra grega:

```
\x -> x ^ 2
```

  - **Não** é uma associação de uma expressão a uma variável
  - É uma expressão que retorna uma função.
  - Como avaliar a função com o parâmetro 2?

Refatorando greaterThan100

```
gt100 :: Integer -> Bool
gt100 x = x > 100


greaterThan100 :: [Integer] -> [Integer]
greaterThan100 xs = filter gt100 xs
```

```
greaterThan100_2 :: [Integer] -> [Integer]
greaterThan100_2 xs =
                    filter (\x -> x > 100) xs
```

Múltiplos Parâmetros

- Funções com mais de um parâmetro em Haskell devem utilizar curring

```
f :: Int -> Int -> Int
f x y = 2*x + y
```

- A razão disso é que todas as funções em Haskell são funções de 1 parâmetro.
- A construção de parâmetros usando currying permite a geração de funções parciais.

```
f' :: Int -> (Int -> Int)
f' x y = 2*x + y
```

- Para funções anônimas podemos utilizar a seguinte notação

```
\x y z -> ...
```

- Essa notação é apenas um açucar sintático – bastante útil – para

```
\x -> (\y -> (\z -> ...)).
```

Exercício - Lab

- Usando funções anônimas, **map**, **filter**, **words** e **unwords.**
  - Importe a biblioteca **Data.Char** e use a função **isUpper** para testar se um caracter é maiúsculo.
- Escreva uma função que receba uma frase descrita em uma string e substitua
  - Todas as ocorrências de siglas por `<sigla>/SIGLA`
  - Todas as ocorrências de nomes próprios por `<nome>/PROPRIO`
  - Exemplo:
    - `f “José foi no DP” => “José/PROPRIO foi no DP/SIGLA”.`

Padrão de Recursão Fold

- **folds** permitem implementar um padrão de recursão comum em listas.
- Considere as três funções abaixo **sum** (soma dos elementos em lista), **product** (produtório dos elementos) e **length** (tamanho da lista).

```
sum' :: [Integer] -> Integer
sum' []     = 0
sum' (x:xs) = x + sum' xs
```

```
product' :: [Integer] -> Integer
product' [] = 1
product' (x:xs) = x * product' xs
```

```
length' :: [a] -> Int
length' []     = 0
length' (_:xs) = 1 + length' xs
```

- O que há em comum entre essas três funções?

Função Fold

- Vamos abstrair as características dessas três funções em uma função de alta ordem chamada **fold** (*dobrar* ou *dobra*).

```
fold :: b -> (a -> b -> b) -> [a] -> b
fold z f []     = z
fold z f (x:xs) = f x (fold z f xs)

fold f z [a,b,c] == a `f` (b `f` (c `f` z))
```

- **fold** está disponível no Prelude com o nome **foldl.**

foldl

- **Fold** → **foldl** mas em ordem levemente diferente

```
foldl :: (a -> b -> b) -> b -> [a] -> b
```

- Reescreva as funções **sum**, **product** e **length** usando a função **foldl.**

```
sum''     = foldl (+) 0
product'' = foldl (*) 1
length''  = foldl (\_ s -> 1 + s) 0
```

- Observem no cartão de referência, onde elas estão descritas como folds especiais (pag. 2, fim da coluna 1)

foldr e foldl

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))

foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

- Qual é a diferença essencial entre **foldl** e **foldr**?
- Implemente as funções **and** e **any** abaixo a partir de **fold**

```
and :: [Bool] -> Bool
l = [2, 4, 4, 6, 10, 1, 7]
and [length l > 3, sum l < 20,
          maximum l == 10, minimum l > 5] => False
```

```
any :: (a -> Bool) -> [a] -> Bool
any (\x -> (length x) > 4)
       ["ricardo","teo","rose","gabriel"] => True
```

Monoids

- Vamos explorar a idéia dos monóides e da generalização das operações de fold em estruturas de dados

Monoids

- Descreva uma tipo de dado árvore binária, considerando os dados armazenados nos nós internos:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
  deriving (Show, Eq)
```

- Operação de criação de uma folha

```
leaf :: a -> Tree a
leaf x = Node Empty x Empty
```

- Escreva uma função que calcule o tamanho (em nós de uma árvore):

```
treeSize :: Tree a -> Integer
treeSize Empty        = 0
treeSize (Node l _ r) = 1 + treeSize l + treeSize r
```

Exercício: Funções em Árvores

- Escreva uma função para calcular a soma dos conteúdos dos nós

- Escreva uma função para calcular a profundidade de uma árvore

- Escreva uma função para planificar uma árvore, gerando uma lista com seus elementos

Padrões nas Operações

- As funções discutidas apresentam padrões em comum:
  - Recebem uma árvore como entrada
  - Caso a árvore seja vazia, retorna uma resposta simples
  - Em cada Nó
    - chama recursivamente em cada ramo
    - combina os resultados das chamadas recursivas com o dado do nó para gerar resultado final.
- Como generalizar as funções anteriores, considerando esses padrões?
- Aspectos que definem o padrão:
  - Tipo de retorno
  - A resposta da função em caso o nó seja vazio
  - Como combinar (a operação de combinação) das chamadas recursivas

Operação fold em Árvore

- Chamaremos nossa generalização de uma operação fold aplicada a uma árvore

```
treeFold :: b -> (b -> a -> b -> b) -> Tree a -> b
treeFold e _ Empty        = e
treeFold e f (Node l x r) = f (treeFold e f l) x
                              (treeFold e f r)
```

- Reimplemente as operações anteriores (`TreeSize`, `treeSum`, `treeDepth` e `flatten`) usando **`treeFold`.**

- treeSize

```
treeSize' :: Tree a -> Integer
treeSize' = treeFold 0 (\l _ r -> 1 + l + r)
```

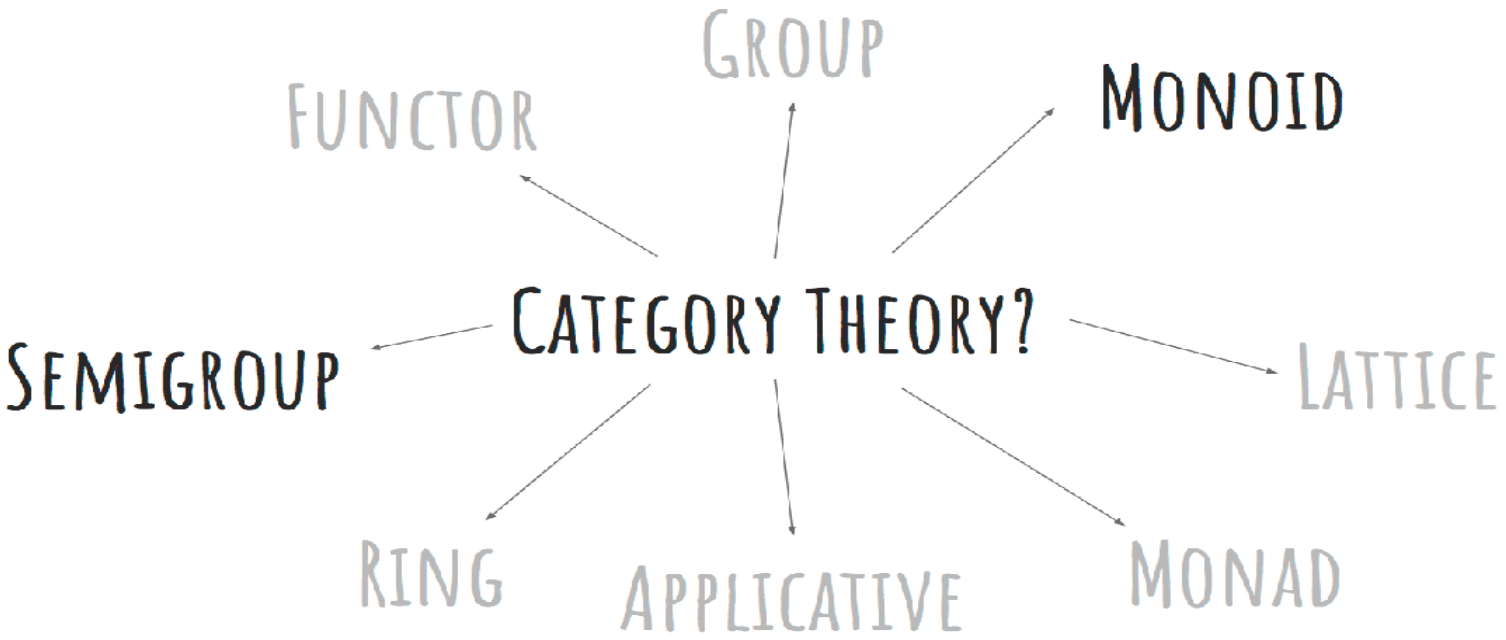
treeFold

- treeSum

- treeDepth

- flatten

Monoides



Um **monoide** pode ser definido de três maneiras completamente equivalentes. Sendo '*' uma operação qualquer:

1. é um conjunto G dotado de uma operação binária para a qual valem as seguintes propriedades:
    1. fechamento: dado $a, b \in G$ o elemento resultante da composição de a e b pertence a G ($a * b \in G$)
    2. associatividade: para todos $a, b, c \in G$ vale $(a * b) * c = a * (b * c) = a * b * c$
    3. existência do elemento neutro: existe um único e tal que para todo $a \in G$ vale $(a * e) = a = (e * a)$

Classe Monoid

- Classe padrão muito útil para resolver diversos problemas em dados, disponível no módulo **Data.Monoid**:

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m

    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

- Para evitar a escrita de **mappend** a todo momento, há o seguinte sinônimo

```
(<>) :: Monoid m => m -> m -> m
(<>) = mappend
```

- Tipos instâncias de Monoid tem
  - Um elemento especial chamado **mempty**
  - Uma operação binária **mappend** (ou **(<>)**), que recebe dois valores e produz um
  - **mempty** deve ser uma identidade para **<>**
  - **<>** deve ser associativa

```
mempty <> x == x
x <> mempty == x
(x <> y) <> z == x <> (y <> z)
a <> b <> c <> d <> e
```

Listas e Monoides

- Listas são monóides do ponto de vista da operação de concatenação

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Ver instâncias de Monoid em:
http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Monoid.html

Exemplos

- Defina monóides sobre as operações de adição e multiplicação em inteiros

```
newtype Sum a = Sum a
  deriving (Eq, Ord, Num, Show)
```

```
getSum :: Sum a -> a
getSum (Sum a) = a
```

```
instance Num a => Monoid (Sum a) where
  mempty  = Sum 0
  mappend = (+)
```

```
newtype Product a = Product a
  deriving (Eq, Ord, Num, Show)
```

```
getProduct :: Product a -> a
getProduct (Product a) = a
```

```
instance Num a => Monoid (Product a) where
  mempty  = Product 1
  mappend = (*)
```

Monoides e Inteiros

- Crie um produtório de uma lista de inteiros **lst** usando **mconcat**

```
lst :: [Integer]
lst = [1,5,8,23,423,99]
```

```
prod :: Integer
prod = getProduct . mconcat . map Product
$ lst
```

Pares de Monóides

- Pares de monoides também formam monoides

```
instance (Monoid a, Monoid b) => Monoid (a,b)
where

  mempty = (mempty, mempty)

  (a,b) `mappend` (c,d) = (a `mappend` c,
                           b `mappend` d)
```