Programação Funcional em Haskell

Funções Genéricas sobre Listas

Maria Adriana Vidigal de Lima

Junho - 2009

Introdução

- Em Haskell, como em qualquer linguagem funcional, funções são objetos de primeira classe, em que funções não avaliadas podem ser passadas como argumentos, construídas ou retornadas como valores de funções.
- Funções que recebem outras funções como argumento, ou retornam uma função como resultado, ou ambos, são chamadas ***Funções de alta ordem***.
- Funções genéricas sobre listas aplicam alguma regra geral sobre os elementos de uma lista. Tais funções são basicamente de três tipos: ***mapeamento***, ***filtragem*** e ***redução***.

Mapeamento

Num **mapeamento**, uma função é aplicada a cada elemento de uma lista, de modo que uma nova lista modificada é retornada.

Motivação ao mapeamento:

Desejamos frequentemente aplicar uma função à uma lista de elementos. Por exemplo, dobrar uma lista de inteiros:

```
dobrar :: [Int] -> [Int]
dobrar [] = []
dobrar (x:xs) = x*2 : dobrar xs

Main> dobrar [1,3,5]
[2,6,10]
```

```
dobro xs = [ x*2 | x <- xs]

Main> dobro [1,3,5]
[2,6,10]
```

Mapeamento: Motivação

Outro exemplo pode ser igualmente definido: adicionar 1 a cada elemento de uma lista de números:

```
incrementa :: (Num a) => [a] -> [a]
incrementa [] = []
incrementa (x:xs) = x+1 : incrementa xs
```

ou simplesmente:

```
incrementar :: (Num a) => [a] -> [a]
incrementar xs = [ x+1 | x <- xs]
```

Mapeamento: Motivação

Na declaração da função incrementar, utilizamos a classe de tipos numéricos *Num* que é uma subclasse de *Eq* e contém seis subclasses e oito tipos numéricos específicos, como *Int*, *Integer*, *Float*, *Double*.

```
incrementar :: (Num a) => [a] -> [a]
incrementar xs = [ x+1 | x <- xs]

Main> incrementa [1,2,3]
[2,3,4]
Main> incrementa [1.1,2.2,3.3]
[2.1,3.2,4.3]
Main> incrementa [1,1.1,2]
[2.0,2.1,3.0]
Main> incrementa [2,'a',3]
ERROR - Cannot infer instance
*** Instance   : Num Char
*** Expression : incrementa [2,'a',3]
```

Mapeamento

Uma função para mapeamento deve receber os seguintes argumentos:

- uma função de transformação
- uma lista de elementos a serem transformados

A linguagem Haskell possui a implementação de uma função de mapeamento, chamada *map*:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

Ou de forma equivalente:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

map :: (a -> b) -> [a] -> [b]

```
                função de entrada          lista de entrada          lista de saída
map :: ( a -> b )          ->     [a]          ->      [b]
map :: (... -> ...)        ->     [...]        ->      [...]
```

As entradas *a* e *b* são de tipos arbitrários e a lista de entrada deve ser do mesmo tipo dos valores aplicados à função. O mesmo ocorre para a lista de saída. Por exemplo:

```
map :: (Int -> Int) -> [Int] -> [Int]
map :: (Char -> Int) -> [Char] -> [Int]
```

Utilizando a função *map*

```
Main> map (+7) [1,2,3]
[8,9,10]

Main> map (even) [1,2,3,4]
[False,True,False,True]

Main> map ("Sr. " ++) ["Joao","Pedro","Luiz"]
["Sr. Joao","Sr. Pedro","Sr. Luiz"]

Main> map (True &&) [True,False]
[True,False]

Main> map (False ||) [False, True]
[False,True]
```

Utilizando a função *map*

Outro exemplo:

```
convertChar xs = [ ord x | x <- xs]
```

```
Main> convertChar "adriana"
[97,100,114,105,97,110,97]
Main> convertChar ['a','b','c']
[97,98,99]
```

De forma equivalente, podemos usar o mapeamento através da função *map*:

```
Main> map ord "adriana"
[97,100,114,105,97,110,97]
Main> map ord ['a','b','c']
[97,98,99]
```

Utilizando a função *map*

A função *map* pode ser usada na definição de outra função, como no exemplo:

```
convStrMaius::[Char] -> [Char]
convStrMaius xs = map (toUpper) xs

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

Main> convStrMaius "abcdef"
"ABCDEF"
```

Exemplo de execução

convStrMaius "abc"
⇒ map (toUpper) [’a’,’b’,’c’]
⇒ (toUpper ’a’) : map (toUpper) [’b’,’c’]
⇒ (toUpper ’a’) : (toUpper ’b’) : map (toUpper) [’c’]
⇒ (toUpper ’a’) : (toUpper ’b’) : (toUpper ’c’) : map (toUpper) [ ]
⇒ (toUpper ’a’) : (toUpper ’b’) : (toUpper ’c’) : [ ]
⇒ (toUpper ’a’) : (toUpper ’b’) : [’C’]
⇒ (toUpper ’a’) : [’B’,’C’]
⇒ [’A’,’B’,’C’]
⇒ "ABC"

Filtragem

- Frequentemente desejamos produzir sub-listas através da seleção de elementos que compartilham uma determinada propriedade.
- A função Haskell `isAlpha::Char -> Bool` decide se um caracter é letra ou não retornando um valor booleano (True ou False), e pode ser vista como uma propriedade de seleção para elementos numa lista.
- Neste contexto, uma função *filtro* recebe a função que define a propriedade e uma lista de entrada, e retorna uma sub-lista contendo os elementos que satisfazem a propriedade.

Definindo o filtro no programa:

Para filtrar as letras numa string, podemos escrever o seguinte código:

```
pegaLetras :: String -> String
pegaLetras  [] = []
pegaLetras  (x:xs)  | isAlpha x = x : pegaLetras  xs
                    | otherwise = pegaLetras  xs

Main> pegaLetras "a1b2c3d4"
"abcd"
```

Definindo o filtro no programa:

Se quisermos apenas os dígitos, podemos utilizar a função `isDigit`:

```
pegaDigitos :: String -> String
pegaDigitos[] = []
pegaDigitos (x:xs) | isDigit x = x : pegaDigitos xs
                   | otherwise = pegaDigitos xs

Main> pegaDigitos "123.324.378-63"
"12332437863"
```

Conhecendo a função `filter`:

Haskell possui uma função de filtragem, denominada `filter` que especifica uma condição a ser aplicada à cada elemento de uma lista, retornando a lista *filtrada*.

```
Main> filter (isDigit) "123-ab4"
"1234"
```

De uma forma geral, podemos definir:

```
p :: a -> Bool
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

Redefinindo a função `pegaDigitos`:

Seja a função original `pegaDigitos`:

```
pegaDigitos :: String -> String
pegaDigitos[] = []
pegaDigitos (x:xs) | isDigit x = x : pegaDigitos xs
                   | otherwise = pegaDigitos xs
```

Podemos reescrever `pegaDigitos` usando a função `filter`:

```
pegaDigitos :: String -> String
pegaDigitos xs = filter isDigit xs
```

Operação de Redução (*folding*)

- Frequentemente desejamos transformar todos os elementos de uma lista num único valor, dada uma propriedade de transformação.
- A função soma pode unir todos os elementos de uma lista numérica: `sum [1,2,3] = 1 + 2 + 3 = 6`.
- Uma **Redução** implementa a operação de aplicar um operador ou função à uma lista de valores e combiná-los.

Operação de Redução (*folding*)

A linguagem Haskell implementa algumas operações de Redução, através das funções `foldr1` e `foldr`. A função

`foldr1` possui dois argumentos: uma função binária sobre um tipo, e uma lista de valores. O resultado é um valor do mesmo tipo da lista de entrada.

```
foldr1 :: (a -> a -> a) -> [a] -> a

Main> foldr1 (+) [1,3,5]
9
Main> foldr1 (*) [1 .. 8]
40320
Main> foldr1 min [4,9,3,5]
3
Main> foldr1 (++) ["Bom"," ","Dia"]
"Bom Dia"
```

Redução (*folding*)

A função `foldr` estende a função foldr1 com mais um parâmetro, um valor *default*. Logo a função recebe uma função a ser aplicada, uma lista de valores e um valor *default*.

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

Main> foldr (+) 1 [1,2,3]
7
Main> foldr (\x -> \y -> y + 1) 0 [5,12,25,14]
4
Main> foldr (++) "ana" ["ab","bc"]
"abbcana"
Main> foldr (&&) False [True,True,True]
False
```

Bibliografia Utilizada

Cláudio Cesar de Sá, Márcio Ferreira da Silva, *Haskell - Uma Abordagem Prática*, Novatec Editora, 2006.