Paradigmas de Programação

Fabrício Olivetti de França

05 de Julho de 2018

Funções de alta ordem

Funções com funções

Vimos anteriormente que o Haskell permite que passemos funções como argumento:

```haskell
duasVezes :: (a -> a) -> a -> a
duasVezes f x = f (f x)
```

Funções com funções

Essas funções são aplicáveis em diversas situações:

```
> duasVezes (*2) 3
12

> duasVezes reverse [1,2,3]
[1,2,3]
```

Funções com funções

Além disso podemos fazer uma aplicação parcial da função, com apenas um argumento, para gerar outras funções:

```
quadruplica = duasVezes (*2)
```

Funções de alta ordem

As funções que recebem uma ou mais funções como argumento, ou que retornam uma função são denominadas **Funções de alta ordem** (*high order functions*).

O uso de funções de alta ordem permitem aumentar a expressividade do Haskell quando confrontamos padrões recorrentes.

Funções de alta ordem para listas

Considere o padrão comum:

```
[f x | x <- xs]
```

que utilizamos para gerar uma lista de números ao quadrado, somar um aos elementos de uma lista, etc.

Map

Podemos definir a função `map` como:

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

Uma função que transforma uma lista do tipo `a` para o tipo `b` utilizando uma função `f :: a -> b`.

Map

Com isso temos uma visão mais clara das transformações feitas em listas:

```
> map (+1) [1,2,3]
[2,3,4]

> map even [1,2,3]
[False, True, False]

> map reverse ["ola", "mundo"]
["alo", "odnum"]
```

Observações sobre o map

1 Ela é um tipo genérico, recebe qualquer tipo de lista 2 Ela pode ser aplicada a ela mesma, ou seja, aplicável em listas de listas:

```
> map (map (+1)) [[1,2],[3,4]]
  => [ map (+1) xs | xs <- [[1,2],[3,4]] ]
  => [ [x+1 | x <- xs] | xs <- [[1,2],[3,4]] ]
```

Map recursivo

Uma definição recursiva de `map` é dada como:

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

Filter

Outro padrão recorrente observado é a filtragem de elementos utilizando guards nas listas:

```
> [x | x <- [1..10], even x]
[2,4,6,8,10]

> [x | x <- [1..10], primo x]
[2,3,5,7]
```

Filter

Podemos definir a função de alta ordem `filter` da seguinte forma:

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

`filter` retorna uma lista de todos os valores cujo o predicado `p` de `x` retorna `True`.

Filter

Reescrevendo os exemplos anteriores:

```
> filter even [1..10]
[2,4,6,8,10]

> filter primo [1..10]
[2,3,5,7]
```

Filter

Podemos passar funções parciais também como argumento:

```
> filter (>5) [1..10]
[6,7,8,9,10]

> filter (/= ' ') "abc def ghi"
"abcdefghi"
```

Filter recursivo

Da mesma forma que `map` podemos definir `filter` recursivamente como:

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

Map e Filter

As duas funções `map` e `filter` costumam serem utilizadas juntas, assim como na compreensão de listas:

```haskell
somaQuadPares :: [Int] -> Int
somaQuadPares ns = sum [n^2 | n <- ns, even n]

somaQuadPares :: [Int] -> Int
somaQuadPares ns = sum (map (^2) (filter even ns))
```

Operador pipe

Podemos utilizar o operador $ para separar as aplicações das funções e remover os parênteses:

```
somaQuadPares :: [Int] -> Int
somaQuadPares ns = sum
                 $ map (^2)
                 $ filter even ns
```

A execução é de baixo para cima.

Outras funções de alta ordem

Outras funções úteis durante o curso:

```
> all even [2,4,6,8]
True

> any odd [2,4,6,8]
False

> takeWhile even [2,4,6,7,8]
[2,4,6]

> dropWile even [2,4,6,7,8]
[7,8]
```

Folding

Função foldr

Vamos recapitular algumas das funções recursivas da aula anterior:

```
sum []     = 0
sum (x:xs) = x + sum xs

product []     = 1
product (x:xs) = x * product xs

length []     = 0
length (_:xs) = 1 + length xs
```

Função `foldr`

Podemos generalizar essas funções da seguinte forma:

```
f [] = v
f (x:xs) = g x (f xs)
```

Função `foldr`

Essa funções é chamada de `foldr`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Função `foldr`

O nome dessa função significa `dobrar`, pois ela justamente dobra a lista aplicando a função `f` em cada elemento da lista e um resultado parcial.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Função `foldr`

Pense nessa lista não-recursivamente a partir da definição de listas:

```
a1 : (a2 : (a3 : []))
```

Função `foldr`

Trocando `:` pela função `f` e `[]` pelo valor `v`:

```
a1 `f` (a2 `f` (a3 `f` v))
```

Função `foldr`

Ou seja:

```
foldr (+) 0 [1,2,3]
```

se torna:

```
1 + (2 + (3 + 0))
```

Função `foldr`

Que é nossa função `sum`:

```
sum = foldr (+) 0
```

Exercício

Defina `product` utilizando `foldr`.

Função `length`

Como podemos implementar `length` utilizando `foldr`?

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

Função `length`

Para a lista:

```
1 : (2 : (3 : []))
```

devemos obter:

```
1 + (1 + (1 + 0))
```

Função `length`

Da assinatura de `foldr`:

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

Percebemos que na função `f` o primeiro argumento é um elemento da lista e o segundo é o valor acumulado.

Função `length`

Dessa forma podemos utilizar a seguinte função anônima:

```
length = foldr (\_ n -> 1+n) 0
```

Exercício (0.5 pto)

Reescreva a função `reverse` utilizando `foldr`:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Folding caudal

Na aula sobre recursão, implementamos muitas dessas funções em sua versão caudal:

```
sum :: Num a => [a] -> a
sum ns = sum' 0 ns
  where
    sum' v []     = v
    sum' v (x:xs) = sum' (v+x) xs
```

Função foldl

Esse padrão é capturado pela função foldl:

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []       = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Função `foldl`

Da mesma forma podemos pensar em `foldl` não recursivamente invertendo a lista:

```
1 : (2 : (3 : []))
  => (([] : 1) : 2) : 3
  => ((0 + 1) + 2) + 3
```

Função `foldl`

Quando `f` é associativo, ou seja, os parênteses não fazem diferença, a aplicação de `foldr` e `foldl` não se altera:

```
sum = foldl (+) 0

product = foldl (*) 1
```

Função `length`

Como ficaria a função `length` utilizando `foldl`?

```
length = foldr (\_ n -> 1+n) 0
length = foldl (??) 0
```

Função `length`

Basta inverter a ordem dos parâmetros:

```
length = foldr (\_ n -> 1+n) 0
length = foldl (\n _ -> n+1) 0
```

Função `reverse`

E a função `reverse`?

O que eu uso? `foldr` ou `foldl`

A escolha entre `foldr` e `foldl`, quando é possível escrever uma função utilizando qualquer um dos dois, é feita após um estudo cuidadoso sobre a performance das duas versões.

Esse tipo de análise será discutida no final do curso.

Exercício

Dada a definição do operador &&:

```haskell
(&&) False _ = False
(&&) _ False = False
(&&) _ _ = True
```

Expanda as seguintes expressões:

```haskell
foldl (&&) False [False, False, False, False]
foldr (&&) False [False, False, False, False]
```

`foldr` vs `foldl`

Uma regra do *dedão* para trabalharmos por enquanto é:

- Se a lista passada como argumento é infinita, use `foldr`
- Se o operador utilizado pode gerar curto-circuito, use `foldr`
- Se a lista é finita e o operador não irá gerar curto-circuito, use `foldl`
- Se faz sentido trabalhar com a lista invertida, use `foldl`

E temos uma função chamada `foldl'` que aprenderemos mais para frente.

Composição de funções

Composição de funções

Na matemática a composição de função $f \circ g$ define uma nova função z tal que $z(x) = f(g(x))$.

No Haskell temos o operador `(.)`:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Composição de funções

Dada uma função que mapeia do tipo `b` para o tipo `c`, e outra que mapeia do tipo `a` para o tipo `b`, gere uma função que mapeie do tipo `a` para o tipo `c`.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Propriedades da composição

A composição de função é associativa:

```
(f . g) . h == f . (g . h)
```

Propriedades da composição

E tem um elemento nulo que é a função `id`:

```
f . id = id . f = f
```

Propriedades da composição

Essas duas propriedades são importantes durante a construção de programas, pois elas permitem o uso do `foldr` (e dentre outras funções de alta ordem):

```
-- cria uma função que é a composição de uma lista de funç
compose :: [a -> a] -> (a -> a)
compose = foldr (.) id
```