Funções de Alta Ordem

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Funções de alta ordem

Funções com funções

UFABC

- 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

UFABC

- Essas funções são aplicáveis em diversas situações:

```
> duasVezes (*2) 3
12

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

Funções com funções

UFABC

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

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

Funções de alta ordem

UFABC

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

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

Funções de alta ordem para listas

UFABC

- Considere o padrão:

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

- Que é muito comum quando queremos gerar uma lista de números ao quadrado, somar um aos elementos de uma lista, etc.

UFABC

- Podemos definir a função `map` como:

```haskell
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`.

UFABC

- 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

UFABC

- `map` funciona para listas genéricas, de qualquer tipo.
- `map` também funciona para qualquer função `f :: a -> b`
- Logo, ela pode ser aplicada a ela mesma, ou seja, aplicável em listas de listas

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

Map recursivo

UFABC

- Uma definição recursiva de `map` é dada como:

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

Filter

UFABC

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

```
1  > [x | x <- [1..10], even x]
2  [2,4,6,8,10]
3
4  > [x | x <- [1..10], primo x]
5  [2,3,5,7]
```

Filter

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

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

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

Filter

UFABC

- Reescrevendo os exemplos anteriores:

```
1  > filter even [1..10]
2  [2,4,6,8,10]
3
4  > filter primo [1..10]
5  [2,3,5,7]
```

UFABC

- Podemos passar funções parciais também como argumento:

```
1  > filter (>5) [1..10]
2  [6,7,8,9,10]
3
4  > filter (/= ' ') "abc def ghi"
5  "abcdefghi"
```

Pergunta

Porque o exemplo acima `(>5)` funciona?

Filter recursivo

UFABC

- 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
```

UFABC

- 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

UFABC

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

```
> :t ($)
($) :: (a -> b) -> a -> b
```

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

- Diferentemente do pipe do Unix, no nosso caso a execução é de baixo para cima.

Outras funções de alta ordem

UFABC

- 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]

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

Folding

Função `foldr`

UFABC

- Vamos recapitular algumas das funções recursivas da aula anterior:

```haskell
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`

UFABC

- Podemos generalizar essas funções da seguinte forma:

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

Função `foldr`

UFABC

- Essa função é chamada de **`foldr`**:

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

Função `foldr`

UFABC

- 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.

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

Função `foldr`

UFABC

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

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

Função `foldr`

UFABC

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

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

Função `foldr`

UFABC

Ou seja:

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

se torna:

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

Função `foldr`

UFABC

Que é nossa função `sum`:

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

Exercício 1

UFABC

- Defina `product` utilizando `foldr`.

Função `length`

UFABC

- Como podemos implementar **`length`** utilizando **`foldr`**?

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

Função `length`

UFABC

Para a lista:

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

devemos obter:

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

Função `length`

UFABC

Da assinatura de `foldr`:

```haskell
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`

UFABC

- Dessa forma podemos utilizar a seguinte função anônima:

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

Pergunta

`length = foldr (+1) 0` funciona?

Exercício 2

UFABC

- Reescreva a função **`reverse`** utilizando **`foldr`**:

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

Folding caudal

UFABC

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

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

Função `foldl`

UFABC

- 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`

UFABC

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

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

Função `foldl`

UFABC

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

```haskell
sum = foldl (+) 0

product = foldl (*) 1
```

Função `length`

UFABC

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

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

Função `length`

UFABC

- Basta inverter a ordem dos parâmetros:

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

Função `reverse`

UFABC

- E a função `reverse`?

Resposta

UFABC

```haskell
1 : (2 : (3 : []))
  => (([] `f` 3) `f` 2) `f` 1

f xs x = ???
reverse = foldl f []
```

Resposta

UFABC

```haskell
1 : (2 : (3 : []))
  => (([] f 3) f 2) f 1

f xs x = x:xs
reverse = foldl f []

-- ou se quiser usar uma lambda
reverse = foldl (\xs x -> x:xs) []
```

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

UFABC

- A escolha entre **`foldr`** e **`foldl`**, quando é possível escrever uma função utilizando qualquer um dos dois, é feita após um estudo cuidadoso sobre o desempenho das duas versões.
- Esse tipo de análise nem sempre é trivial e será discutida no final do curso.
- Por enquanto, utilizaremos um exemplo para mostrar a razão desta discussão.

Exercício 3

UFABC

- 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`

UFABC

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`

- Bônus! E temos ainda uma outra função chamada `foldl'` que aprenderemos mais para frente.

Composição de funções

Composição de funções

UFABC

- 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 `(.)`:

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

Composição de funções

UFABC

- A assinatura da função `(.)` merece um pouco mais de atenção
- 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`.

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

Propriedades da composição

UFABC

- A composição de função é associativa:

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

Propriedades da composição

UFABC

- Temos também um elemento neutro que é a função `id`:

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

Propriedades da composição

UFABC

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

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

Exemplo do uso de composição

UFABC

```haskell
-- A função fn
fn x = ceiling (negate (tan (cos (max 50 x))))
-- Pode ser reescrita
fn = ceiling . negate . tan . cos . max 50
```

Exemplo do uso de composição

UFABC

```haskell
somaDosQuadradosImpares :: Integer
somaDosQuadradosImpares =
  sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
-- ou
somaDosQuadradosImpares =
  sum . takeWhile (<10000) . filter odd . map (^2) $
  ↪ [1..]
-- ou
somaDosQuadradosImpares =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (<10000) oddSquares
    in  sum belowLimit
```

- Quando em dúvida sobre qual usar se o desempenho for o mesmo opte pela versão mais legível.

($) *vs.* (.)

UFABC

```
> :t ($)
($) :: (a -> b) -> a -> b
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- Observe o código abaixo

```
f x = sin $ abs x
g = sin . abs
h x = (sin . abs) x
i x = sin . abs $ x
```

Pergunta

Quais são os tipos de `f`, `g`, `h` e `i`?