

Programação Funcional

6ª Aula — Funções de ordem superior

Sandra Alves
DCC/FCUP

2018/19

Funções de ordem superior

Uma função é de *ordem superior* se tem um argumento que é uma função ou um resultado que é uma função.

Exemplo: o primeiro argumento de *twice* é uma função.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Porquê ordem superior?

- Permite definir *padrões de computação* comuns que podem ser facilmente re-utilizados.
- Facilita a definição de *bibliotecas para domínios específicos*:
 - processamento de listas;
 - formatação de texto;
 - “parsing”;
 - ...
- Podemos provar *propriedades gerais* das funções de ordem superior que são válidas em qualquer use específico.

A função *map*

A função *map* aplica uma função a cada elemento numa lista.

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

Exemplos:

```
> map (+1) [1,3,5,7]
[2,4,6,8]
```

```
> map isLower "Hello!"
[False,True,True,True,True,False]
```

Podemos definir *map* usando uma lista em compreensão:

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

Também podemos definir *map* usando recursão:

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

Esta forma será útil para provar propriedades usando indução.

Função *filter*

A função *filter* seleciona os elementos duma lista que satisfazem um *predicado* (isto é, uma função cujo resultado é um valor booleano).

```
filter :: (a -> Bool) -> [a] -> [a]
```

Exemplos:

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

> filter isLower "Hello, world!"
"elloworld"
```

Podemos definir *filter* usando uma lista em compreensão:

```
filter p xs = [x | x<-xs, p x]
```

Também podemos definir *filter* usando recursão:

```
filter p [] = []
filter p (x:xs)
  | p x      = x : filter p xs
  | otherwise = filter p xs
```

Funções *takeWhile* e *dropWhile*

takeWhile seleciona o maior prefixo numa lista cujos elementos verificam um predicado.

dropWhile remove o maior prefixo cujos elementos verificam um predicado.

As duas funções têm o mesmo tipo:

```
takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
```

Exemplos:

```
> takeWhile isLetter "Hello, world!"  
"Hello"
```

```
> dropWhile isLetter "Hello, world!"  
", world!"
```

```
> takeWhile (\n -> n*n<10) [1..5]  
[1,2,3]
```

```
> dropWhile (\n -> n*n<10) [1..5]  
[4,5]
```

Definições recursivas de *takeWhile* e *dropWhile* (do prelúdio-padrão):

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

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

As funções *all* e *any*

all verifica se um predicado é verdadeiro para *todos* os elementos numa lista.

any verifica se um predicado é verdadeiro para *algum* elemento numa lista.

As duas funções têm o mesmo tipo:

```
all, any :: (a -> Bool) -> [a] -> Bool
```

Exemplos:

```

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

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

> all isLower "Hello, world!"
False

> any isLower "Hello, world!"
True

```

Podemos definir *all* e *any* usando *map*, *and* e *or*:

```

all p xs = and (map p xs)
any p xs = or (map p xs)

```

A função *foldr*

Muitas funções sobre listas seguem o seguinte padrão de definição recursiva:

```

f []      = z
f (x:xs) = x  $\oplus$  f xs

```

Ou seja, f transforma:

a lista vazia em z ;

a lista não-vazia $x : xs$ usando uma operação \oplus para combinar x com $f\ xs$.

Exemplos:

```

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

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

and []       = True
and (x:xs) = x && and xs

or []        = False
or (x:xs) = x || or xs

length []    = 0
length (x:xs) = 1 + length xs

```

Exemplos:

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

$z = 0$
 $\oplus = +$

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

```
and [] = True
and (x:xs) = x && and xs
```

```
or [] = False
or (x:xs) = x || or xs
```

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

Exemplos:

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

$z = 0$
 $\oplus = +$

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

$z = 1$
 $\oplus = *$

```
and [] = True
and (x:xs) = x && and xs
```

```
or [] = False
or (x:xs) = x || or xs
```

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

Exemplos:

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

$z = 0$
 $\oplus = +$

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

$z = 1$
 $\oplus = *$

```
and [] = True
and (x:xs) = x && and xs
```

$z = \text{True}$
 $\oplus = \text{True}$

```
or [] = False
or (x:xs) = x || or xs
```

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

Exemplos:

sum [] = 0	$z = 0$
sum (x:xs) = x + sum xs	$\oplus = +$
product [] = 1	$z = 1$
product (x:xs) = x * product xs	$\oplus = *$
and [] = True	$z = \text{True}$
and (x:xs) = x && and xs	$\oplus = \&\&$
or [] = False	$z = \text{False}$
or (x:xs) = x or xs	$\oplus = $
length [] = 0	
length (x:xs) = 1 + length xs	

Exemplos:

sum [] = 0	$z = 0$
sum (x:xs) = x + sum xs	$\oplus = +$
product [] = 1	$z = 1$
product (x:xs) = x * product xs	$\oplus = *$
and [] = True	$z = \text{True}$
and (x:xs) = x && and xs	$\oplus = \&\&$
or [] = False	$z = \text{False}$
or (x:xs) = x or xs	$\oplus = $
length [] = 0	$z = 0$
length (x:xs) = 1 + length xs	$\oplus = \backslash x \ n \rightarrow 1 + n$

A função de ordem superior *foldr* (“fold right”) abstrai este padrão de recursão; os seus argumentos são a operação \oplus e o valor z :

```
sum      = foldr (+) 0

product = foldr (*) 1

and      = foldr (&&) True

or       = foldr (||) False

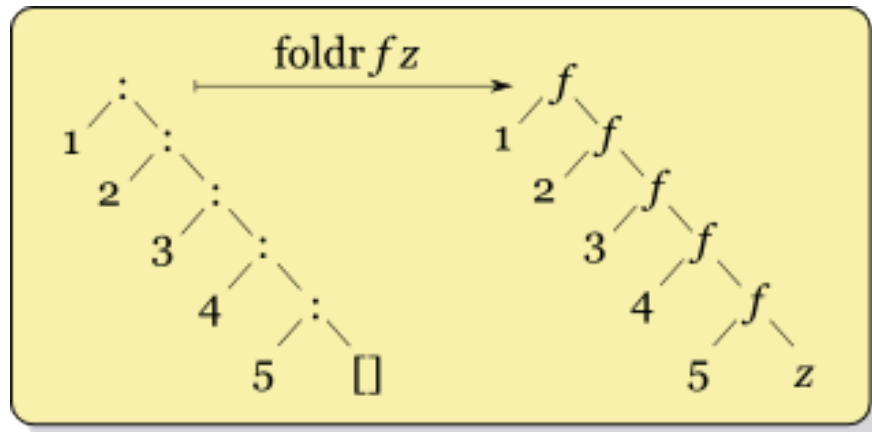
length  = foldr (\x n->n+1) 0
```

Definição recursiva de *foldr* (do prelúdio-padrão):

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

Podemos visualizar $\text{foldr } f \ z$ como a transformação que substitui:

- cada $(:)$ por f ;
- $[]$ por z .



Exemplo:

```
sum [1,2,3,4]
=
foldr (+) 0 [1,2,3,4]
=
foldr (+) 0 (1:(2:(3:(4:[]))))
=
1+(2+(3+(4+0)))
=
10
```

Outro exemplo:

```
product [1,2,3,4]
=
foldr (*) 1 [1,2,3,4]
=
foldr (*) 1 (1:(2:(3:(4:[]))))
=
1*(2*(3*(4*1)))
=
24
```

A função *foldl*

A função *foldr* transforma uma lista usando uma operação associada à direita (“fold right”):

$$\text{foldr } (\oplus) \ v \ [x_1, x_2, \dots, x_n] = x_1 \oplus (x_2 \oplus (\dots (x_n \oplus v) \dots))$$

Existe outra função *foldl* que transforma uma lista usando uma operação associada à esquerda (“fold left”):

$$\text{foldl } (\oplus) \ v \ [x_1, x_2, \dots, x_n] = ((\dots ((v \oplus x_1) \oplus x_2) \dots) \oplus x_n)$$

Se f for *associativa* com *elemento neutro* z , então $foldr\ f\ z$ e $foldl\ f\ z$ dão o mesmo resultado.

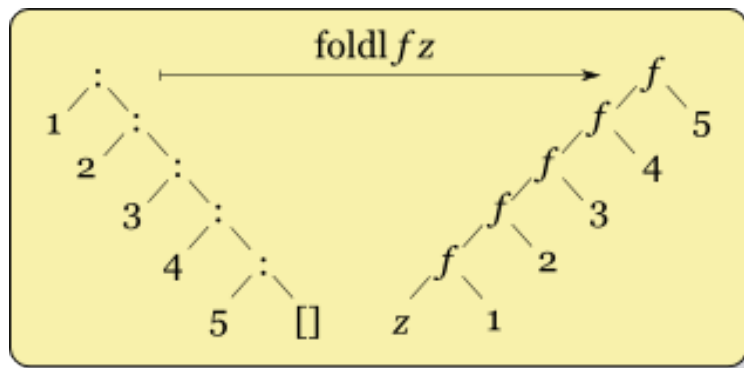
```
foldl (+) 0 [1,2,3,4]
=
(((0+1)+2)+3)+4
=
10
```

```
foldr (+) 0 [1,2,3,4]
=
1+(2+(3+(4+0)))
=
10
```

Tal como *foldr*, a função *foldl* está definida no prelúdio-padrão usando recursão:

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

No entanto, pode ser mais fácil visualizar *foldl* como uma transformação sobre listas:



Fonte: [http://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](http://en.wikipedia.org/wiki/Fold_(higher-order_function)).

Outras funções de ordem superior

A função (\cdot) é a *composição* de duas funções.

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

Exemplo

```
par :: Int -> Bool
par x = x `mod` 2 == 0
```

```
impar :: Int -> Bool
impar = not . par
```


A composição permite muitas vezes simplificar definições embricadas, omitido os parêntesis e o argumento.

Exemplo:

```
f xs = sum (map (^2) (filter par xs))
```

é equivalente a

```
f = sum . map (^2) . filter par
```