Programação Funcional
Aula 7 — Funções de ordem superior

Pedro Vasconcelos
DCC/FCUP

2021

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 *map* é uma função, logo *map* é uma função de ordem superior.

```
> map (^2) [1,2,3,4]
[1,4,9,16]
```

Porquê ordem superior?

- Permite parametrizar funções passando-lhes *operações* e não apenas *dados*
- Permite definir padrões de computação comuns que podem ser facilmente re-utilizados
- Mais tarde veremos que podemos provar propriedades gerais de funções de ordem superior
  - exemplo: *map* mantém o comprimento da lista dada

Nesta aula

Algumas funções de ordem superior definidas no prelúdio-padrão:

- `map`
- `filter`
- `takeWhile`, `dropWhile`
- `all`, `any`
- `foldr`, `foldl`
- `(.)` (composição)

A função *map*

A função *map* aplica uma função a cada elemento duma 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]
```

A função *map* (cont.)

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
```

(A forma recursiva será útil quando provarmos propriedades usando indução.)

Função *filter*

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

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

Exemplos

```
> filter (\n->n`mod`2==0) [1..10]
[2,4,6,8,10]

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

Função *filter* (cont.)

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 duma 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]
```

Funções *takeWhile* e *dropWhile* (cont.)

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]
```

Funções *takeWhile* e *dropWhile* (cont.)

Poderiamos definir*takeWhile* e *dropWhile* de forma recursiva.

```
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 duma lista.

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

As duas funções têm o mesmo tipo:

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

As funções *all* e *any* (cont.)

Exemplos

```
> all (\n -> n`mod`2==0) [2,4,6,8]
True

> any (\n -> n`mod`2/=0) [2,4,6,8]
False

> all isLower "Hello, world!"
False

> any isLower "Hello, world!"
True
```

As funções *all* e *any* (cont.)

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

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

Também podemos definir por recursão:

```
all p []     = True
all p (x:xs) = p x && all p xs

any p []     = False
any p (x:xs) = p x || any p xs
```

A função *foldr*

Muitas transformações sobre listas seguem o seguinte padrão de *recursão primitiva*:

$$
\begin{aligned}
&\texttt{f []} && = z\\
&\texttt{f (x:xs)} && = \texttt{x} \oplus \texttt{f xs}
\end{aligned}
$$

Ou seja, *f* transforma:

a lista vazia em z;

a lista não-vazia $x : xs$ usando uma operação $\oplus$ para combinar x com o resultado da função para xs.

A função *foldr* (cont.)

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
```

$z = \texttt{True}$
$\oplus = \texttt{\&\&}$

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

$z = \texttt{False}$
$\oplus = \texttt{||}$

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

$z = 0$
$\oplus = \backslash\_\ n \rightarrow 1 + n$

A função *foldr* (cont.)

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 (\_ n->n+1) 0
```

A função *foldr* (cont.)

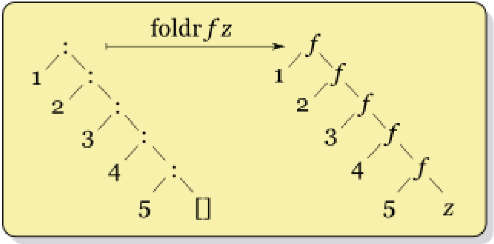
A definição recursiva de *foldr* (do prelúdio-padrão) exprime o padrão de recursão.

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

A função *foldr* (cont.)

É possível visualizar *foldr f z* como uma transformação sobre estruturas de listas:

- cada (:) transforma-se numa aplicação de *f*;
- [] transforma-se na constante *z*.




```
        foldr f z [1,2,3,4,5]
=       foldr f z (1:2:3:4:5:[])
=     f 1 (f 2 (f 3 (f 4 (f 5 z))))
```

A função *foldr* (cont.)

Exemplo

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

A função *foldr* (cont.)

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*):

$$\textit{foldr}\ (\oplus)\ z\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus z) \ldots))$$

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

$$\textit{foldl}\ (\oplus)\ z\ [x_1, x_2, \ldots, x_n] = ((\ldots ((z \oplus x_1) \oplus x_2) \ldots) \oplus x_n)$$

A função *foldl* (cont.)

Se *f* for *associativa* e *z* for o *elemento neutro*, 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
```

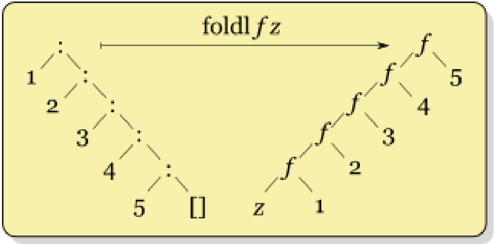
A função *foldl* (cont.)

Tal como *foldr*, a função *foldl* pode ser definida por recursão:

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

A função *foldl* (cont.)

Pode ser mais fácil visualizar *foldl* como uma transformação sobre listas.




```
      foldl f z [1,2,3,4,5]
=     foldl f z (1:2:3:4:5:[])
= f (f (f (f (f z 1) 2) 3) 4) 5
```

Composição

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
```

Composição (cont.)

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

Exemplo

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

é equivalente a

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