

Haskell: Generalização

UFRN, 2018

Funções como argumentos

- Além de números, tuplas e listas, funções também podem ser argumentos para outras funções.
- Funções que recebem outras funções como argumentos são chamadas funções de alta ordem.

Motivação

- Função para retornar uma lista com os ***dobros*** dos valores dos elementos da lista original:

```
1 double :: [Int] -> [Int]
2 double [] = []
3 double (a:x) = (2*a):double x
```

Motivação

- Função para retornar uma lista com os *triplos* dos valores dos elementos da lista original:

```
1 treble :: [Int] -> [Int]
2 treble [] = []
3 treble (a:x) = (3*a):treble x
```

Motivação

```
1 double :: [Int] -> [Int]
2 double [] = []
3 double (a:x) = (2*a) : double x
```

```
1 treble :: [Int] -> [Int]
2 treble [] = []
3 treble (a:x) = (3*a) : treble x
```

- O padrão destas definições é o mesmo, muda apenas o modo como os elementos da lista são transformados.

Generalização

- Uma função genérica para transformar listas de inteiros deveria então ter **dois** argumentos:
 - a lista a ser transformada.
 - uma função que especifica a transformação a ser realizada em cada elemento da lista;
- Exemplos de funções de transformação:

```
1 times2, times3 :: Int -> Int
2 times2 n = 2*n
3 times3 n = 3*n
```

Generalização

- Função genérica para transformar listas de inteiros:

```
1 mapInt f [] = []  
2 mapInt f (a:x) = (f a):mapInt f x
```

- A função `mapInt` recebe como argumentos:
 - uma função `f` que especifica a transformação a ser realizada em cada elemento da lista;
 - a lista a ser transformada.

Generalização

- Função genérica para transformar listas de inteiros:

```
1 mapInt f [] = []  
2 mapInt f (a:x) = (f a) : mapInt f x
```

- Outra solução - usando *list comprehensions*:

```
1 mapInt f l = [ (f a) | a <- l]
```


Exemplo de Execução

```
1 mapInt f [] = []  
2 mapInt f (a:x) = (f a) : mapInt f x  
3  
4 times2 n = 2*n
```

```
MapInt times2 [1,4]  
= (times2 1) : mapInt times2 [4]  
= 2 : mapInt times2 [4]  
= 2 : ((times2 4) : mapInt times2 [])  
= 2 : (8 : mapInt times2 [])  
= 2 : (8 : [])  
= [2,8]
```

Exemplo de Execução

```
1 mapInt f [] = []  
2 mapInt f (a:x) = (f a) : mapInt f x  
3  
4 times3 n = 3*n
```

```
MapInt times3 [1,4]  
= (times3 1) : mapInt times3 [4]  
= 3 : mapInt times3 [4]  
= 3 : ((times3 4) : mapInt times3 [])  
= 3 : (12 : mapInt times3 [])  
= 3 : (12 : [])  
= [3,12]
```

Redefinindo double e treble

```
1 mapInt f [] = []
2 mapInt f (a:x) = (f a) : mapInt f x
3
4 times2, times3 :: Int -> Int
5 times2 n = 2*n
6 times3 n = 3*n
7
8 double l = mapInt times2 l
9 treble l = mapInt times3 l
```

```
Main> double [1,4]
[2,8]
```

```
Main> treble [1,4]
[3,12]
```

Definindo tipos

```
1 mapInt :: t1??? -> t2??? -> t3???  
2 mapInt f [] = []  
3 mapInt f (a:x) = (f a) : mapInt f x
```

- Qual o tipo de MapInt ?
 - o primeiro argumento é uma função que transforma um inteiro em outro inteiro;
 - o segundo argumento é uma lista de inteiros;
 - retorna outra lista de inteiros.

Definindo tipos

```
1 mapInt :: (Int->Int) -> [Int] -> [Int]
2 mapInt f [] = []
3 mapInt f (a:x) = (f a) : mapInt f x
```

- Qual o tipo de MapInt ?
 - o primeiro argumento é uma função que transforma um inteiro em outro inteiro;
 - o segundo argumento é uma lista de inteiros;
 - retorna outra lista de inteiros.

Vantagens

- Maior clareza e simplicidade nas definições.
- Facilidade de modificação.
- Reutilização de definições.
 - MapInt pode ser usada em muitas outras situações.

Exemplo 1

```
1 sales :: Int -> Int
2 sales x
3   | x == 0      = 12
4   | x == 1      = 20
5   | x == 2      = 18
6   | x == 3      = 25
7   | otherwise   = 0
```

Função totalSales
usa uma definição
específica para sales.

```
8 totalSales :: Int -> Int
9 totalSales n
10  | n == 0      = sales 0
11  | otherwise   = totalSales (n-1) + sales n
```

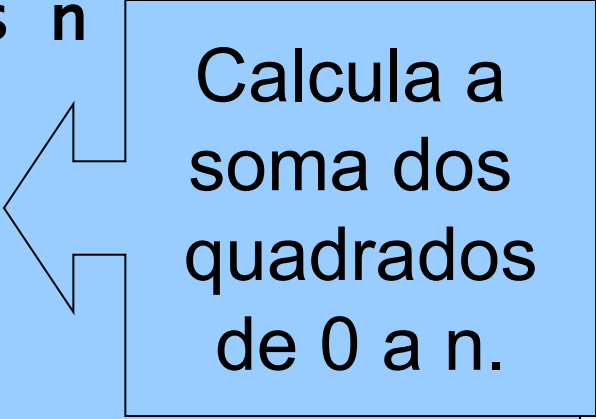
Exemplo 1 - Generalização

```
8 total :: (Int -> Int) -> Int -> Int
9 total f n
10   | n==0      = f 0
11   | otherwise = total f (n-1) + f n
12
13 totalSales n = total sales n
```

- A função `total` pode ser usada em muitas outras situações...

Exemplo 1 - Generalização

```
8 total :: (Int -> Int) -> Int -> Int
9 total f n
10   | n==0      = f 0
11   | otherwise = total f (n-1) + f n
12
13 totalSales n = total sales n
14
15 sumSquares :: Int -> Int
16 sumSquares n = total sq n
17
18 sq :: Int -> Int
19 sq x = x*x
```



Calcula a soma dos quadrados de 0 a n.

Exemplo 2

- Soma dos elementos de uma lista de inteiros:

$e1 + e2 + \dots + ek$

- Máximo elemento de uma lista de inteiros:

$e1 \text{ 'maxi' } e2 \text{ 'maxi' } \dots \text{ 'maxi' } ek$

- Tipos das funções aplicadas a **pares** de elementos inteiros:

$(+), \text{ 'maxi' } :: \text{ Int } \rightarrow \text{ Int } \rightarrow \text{ Int }$

Exemplo 2 - Generalização

```
1 foldInt :: (Int -> Int -> Int) -> [Int] -> Int
2
3 foldInt f [a]      = a
4 foldInt f (a:b:x) = f a (foldInt f (b:x))
5
6
7 sumList l = foldInt (+) 1
8 maxList l = foldInt maxi 1
```

Exemplo 3

- Filtrar algarismos de um texto:
`digits "29 February 1996"`
`= "291996"`
- Filtrar letras de um texto:
`letters "29 February 1996"`
`= "February"`
- Tipos das funções aplicadas a cada elemento:
`isDigit, isLetter :: Char -> Bool`

Exemplo 3 - Generalização

```
1  filterString :: (Char -> Bool) -> [Char] -> [Char]
2
3  filterString p [] = []
4  filterString p (a:x)
5      | p a          = a : filterString p x
6      | otherwise    = filterString p x
7
8  isDigit, isLetter :: Char -> Bool
9  isDigit ch = ('0'<=ch && ch<='9')
10 isLetter ch = ('a'<=ch && ch <='z') ||
11              ('A'<=ch && ch <='Z')
12
13 digits st = filterString isDigit st
14 letters st = filterString isLetter st
```

Exemplo 3 – Solução Alternativa

```
1  filterString :: (Char -> Bool) -> [Char] -> [Char]
2
3  filterString p x = [ a | a<-x , p a ]
4
5  isDigit, isLetter :: Char -> Bool
6  isDigit ch = ('0'<=ch && ch<='9')
7  isLetter ch = ('a'<=ch && ch <='z') ||
8                ('A'<=ch && ch <='Z')
9
10 digits st = filterString isDigit st
11 letters st = filterString isLetter st
```

Polimorfismo

- Função que calcula o comprimento de uma lista:

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

- Qual o tipo de `length` ?

`length :: [t] -> Int`

`t = variável de tipo`

Exemplos de Polimorfismo

- Função que calcula o reverso de uma lista:

```
1 rev [] = []  
2 rev (a:x) = rev x ++ [a]
```

- Qual o tipo de `rev` ?

`rev :: [t] -> [t]`



Importante: retorna lista do mesmo tipo
que o argumento de entrada

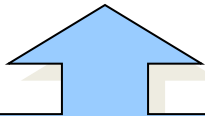
Exemplos de Polimorfismo

- Função para construir lista de pares:

```
1 zip (a:x) (b:y) = (a,b):zip x y
2 zip _ _ = []
```

- Qual o tipo de zip ?

`zip :: [t] -> [u] -> [(t,u)]`



Importante: os tipos `t` e `u`
não estão relacionados!

Polimorfismo - Vantagens

- Definições genéricas.
- Reutilização em muitas situações diferentes.
 - Por exemplo, a função `length` calcula o comprimento de listas de qualquer tipo.
 - Sem polimorfismo, seria necessário criar uma versão de `length` para cada tipo de lista a ser utilizada.

Polimorfismo e Ordem Mais Alta

- Importantes funções que são polimórficas e de ordem mais alta, simultaneamente:
 - **map** : uma lista L2 é criada a partir de uma lista L1, aplicando uma operação sobre cada elemento de L1.
 - **fold** : um valor é calculado, resultado da aplicação de uma operação binária ao longo de toda uma lista de elementos.

Polimorfismo e Ordem Mais Alta

- **filter** : uma lista L2 é criada a partir de uma lista L1, selecionando alguns elementos de L1 que satisfazem a uma determinada propriedade.
- As funções `map`, `fold` e `filter` não impõem qualquer restrição sobre os tipos envolvidos.

Map

- Uma lista L2 é criada a partir de uma lista L1, aplicando uma operação sobre cada elemento de L1.
- Não há restrição sobre o tipo dos elementos de L1, que ainda pode ser diferente do tipo dos elementos de L2.

$$\begin{array}{ccccccc} L1 = [& e1 & , & e2 & , & \dots & , & ek &] \\ & \downarrow f & & \downarrow f & & & & \downarrow f & \\ L2 = [& f(e1) & , & f(e2) & , & \dots & , & f(ek) &] \end{array}$$

Map

```
1 times2, times3 :: Int -> Int
2 times2 n = 2*n
3 times3 n = 3*n
```

```
Main> map times2 [2,3,4]
[4,6,8]
Main> map times3 [2,3,4]
[6,9,12]
```

Map

```
1 isEven :: Int -> Bool  
2 isEven n = (n `mod` 2 == 0)
```

```
Main> map isEven [2,3,4]  
[True,False,True]
```

Map

```
1 offset = ord 'a' - ord 'A'
2 isCapital ch = (ch >= 'A') && (ch <= 'Z')
3 small :: Char -> Char
4 small ch
5     | isCapital ch      = chr (ord ch + offset)
6     | otherwise         = ch
```

```
Main> map small "ABC Futebol Clube"
"abc futebol clube"
```


Map

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

```
Main> map length ["ABC", "Futebol", "Clube"]
[3,7,5]
Main> map length [2,3,4]
ERRO DE TIPO!!!
```

Map

- Definição:

```
1 map :: (t -> u) -> [t] -> [u]
2 map f [] = []
3 map f (a:x) = f a : map f x
```

- Definição alternativa:

```
1 map :: (t -> u) -> [t] -> [u]
2 map f l = [ f a | a <- l ]
```

Fold

- Um valor é calculado, resultado da aplicação de uma operação binária ao longo de toda uma lista de elementos.
- Não há restrição sobre o tipo dos elementos da lista.

`fold f [e1] = e1`

`fold f [e1 , e2 , ... , ek]
= e1 `f` (e2 `f` (... `f` ek) ...)
= f e1 (fold f [e2 , ... , ek])`

Fold

```
Main> fold (+) [1..3]
```

```
6
```

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

```
True
```

```
Main> fold (++) ["ABC"," Fut", " ", "Clube"]  
"ABC Fut Clube"
```

```
Main> fold (-) [1..3]
```

```
2
```

Fold

```
1 maxi :: Int -> Int -> Int
2 maxi a b
3   | a > b      = a
4   | otherwise  = b
```

```
Main> fold maxi [2,4,3]
4
Main> fold maxi [2]
2
Main> fold maxi []
ERRO!!!
```

Fold

- Definição:

```
1 fold :: (t -> t -> t) -> [t] -> t
2 fold f [a] = a
3 fold f (a:b:x) = f a (fold f (b:x))
```

- Os tipos da definição podem ser mais gerais?
- Qual a restrição sobre o tipo da operação f que será executada ao longo da lista?

Fold

- Supondo uma lista $[e1, \dots, ek]$ de tipo $[t]$:

```
fold f [ e1 , e2 , ... , ek ]  
  = e1 `f` (e2 `f` (... `f` ek) ...)  
  = f e1 (fold f [ e2 , ... , ek ])
```

tipo do primeiro
argumento de **f**
tem que ser **t**

tipo do segundo
argumento de **f** tem que
ser o mesmo do valor de
retorno de **fold**

Fold

- Supondo uma lista $[e1, \dots, ek]$ de tipo $[t]$:

```
fold  f  [ e1 , e2, e3 ]  
      = f e1 (fold f [ e2 , e3 ])  
      = f e1 (f e2 (fold f [ e3 ]))  
      = f e1 (f e2 e3)
```



tipo do valor de retorno de **f** deve ser o mesmo tipo do segundo argumento de **f**

Fold

- Restrições:
 - tipo do primeiro argumento de **f** tem que ser **t**, o tipo dos elementos da lista.
 - tipo do segundo argumento de **f** tem que ser o mesmo tipo do valor de retorno de **fold**.
 - tipo do valor de retorno de **f** deve ser o mesmo tipo do segundo argumento de **f**.

```
1 fold :: (t -> u -> u) -> [t] -> u
2 fold f [a] = a
3 fold f (a:b:x) = f a (fold f (b:x))
```

Fold

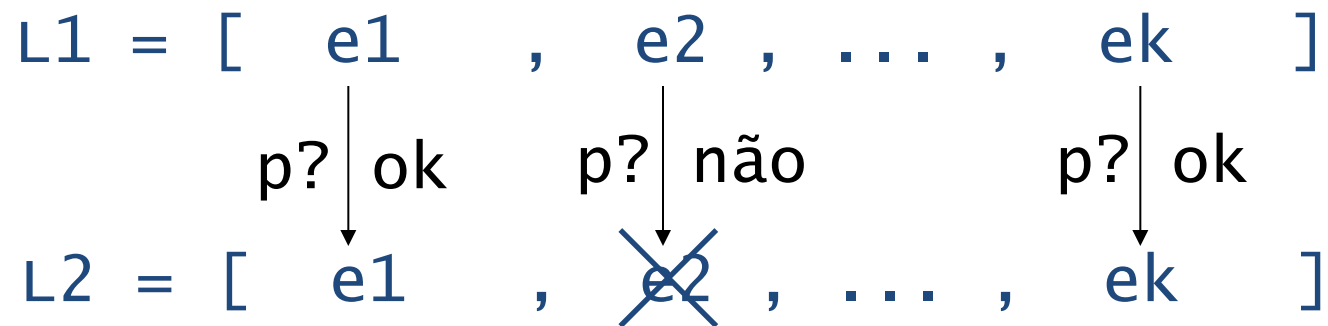
- Acrescentando um argumento a `fold`:

```
1 fold1 :: (t -> u -> u) -> u -> [t] -> u
2 fold1 f s [] = s
3 fold1 f s (a:x) = f a (fold1 f s x)
4
5 sumListInt l = fold1 (+) 0 l
```

```
Main> sumListInt [1..3]
6
Main> sumListInt [3]
3
Main> sumListInt []
0
```

Filter

- Uma lista L2 é criada a partir de uma lista L1, selecionando alguns elementos de L1 que satisfazem a uma determinada propriedade.
- Não há restrição sobre o tipo dos elementos de L1, mas é o mesmo tipo dos elementos de L2.



Filter

- Supondo uma lista $[e_1, \dots, e_k]$ de tipo $[t]$, a propriedade p que é aplicada aos elementos da lista é do tipo:

$t \rightarrow \text{Bool}$

$L1 = [e_1 , e_2 , \dots , e_k]$

$\begin{array}{ccc} p? \downarrow \text{ok} & p? \downarrow \text{não} & p? \downarrow \text{ok} \\ L2 = [e_1 , \cancel{e_2} , \dots , e_k] \end{array}$

Filter

```
1 isEven :: Int -> Bool  
2 isEven n = (n `mod` 2 == 0)
```

```
Main> filter isEven [1..6]  
[2,4,6]
```

Filter

```
1 nonEmpty :: String -> Bool
2 nonEmpty st = (st /= "")
```

```
Main> filter nonEmpty ["ABC","", "Fut Clube"]
["ABC", "Fut Clube"]
```

Filter

- Definição:

```
1 filter :: (t -> Bool) -> [t] -> [t]
2 filter p [] = []
3 filter p (a:x)
4   | p a      = a : filter p x
5   | otherwise = filter p x
```

- Definição alternativa:

```
1 filter :: (t -> Bool) -> [t] -> [t]
2 filter p l = [ a | a <- l, p a ]
```