Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Polimorfismo, Sobrecarga e Generalização

(COMP0393)

Aula 11

dcomp
departamento de computação

Funções Monomórficas

- Considere a função:

```
ehMinusculo c = ('a' <= c) && (c <= 'z')
```

- O argumento de `ehMinusculo` precisa ser um caracter senão a comparação de `c` com os caracteres `'a` e `'z` não poderiam ser executadas.
- Dizemos que `ehMinusculo` é uma função **monomórfica**, pois ela se aplica somente a um único tipo

```
ehMinusculo::Char->Bool
ehMinusculo c = ('a' <= c) && (c <= 'z')
```

Polimorfismo Paramétrico

- Mas:

```
fst (x,y)= x
```

- Não há nenhuma restrição acerca do tipo dos componentes da tupla. Eles podem ser de qualquer tipo!
- Neste caso temos uma definição **genérica**. A função atua sobre uma **família de tipos**.
- Dizemos que `fst` é uma função **polimórfica** e o tipo da função é um **politipo.**
- A função `fst` se aplica a valores cujo tipo tem a forma `(t,r)` e retorna um valor de tipo `t`, onde `t` e `r` são tipos quaisquer

```
fst :: (t,r)-> t
fst (x,y) = x
```

```
fst (3,6) ↝ 3
fst ("ab", 1) ↝ "ab"
fst ((3,'a'), "abcd") ↝(3,'a')
```

Mais exemplos..

```
snd :: (t,r)-> r
snd (x,y) = y

ident :: t -> t
ident x = x
```

```
ident [3,6] ↝ [3,6]
ident [[],[1,2]]↝ [[],[1,2]]
ident (3,'a')↝(3,'a')
```

A vasta maioria da funções pré-definidas de Haskell são polimórficas!

```
head :: [t] -> t
head (x:xs) = x
```

```
head [3,6] ↝ 3
head ["Maria", "Pedro"] ↝ "Maria"
head [(3,'a'),(2, 'd')] ↝(3,'a')
```

```
tail :: [t] -> [t]
tail (x:xs) = xs
```

```
tail [3,6] ↝ [6]
tail ["Maria", "Pedro"]↝ ["Pedro"]
tail [(3,'a'),(2, 'd')] ↝[(2,'d')]
```

Polimorfismo Paramétrico

- Polimorfismo permite **generalização** no sentido de um mesmo programa pode ser usado para inúmeros tipos.
- Quais são os tipos genéricos das funções abaixo?

```
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

drop 0 xs = xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs

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

zip [] ys = []
zip xs [] =
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Polimorfismo Paramétrico

- Quais são os tipos genéricos das funções abaixo?

```
take :: Int -> [a] -> [a]
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs

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

zip :: [a]->[b]->[(a,b)]
zip [] ys = []
zip xs [] =
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Sobrecarga

- Um mesmo nome para diferentes entidades
- O nome (+) usamos para a soma de Int, Integer, Float, Double, ...
- O nome div para a divisão inteira de números Int e números Integer
- Sozinha, é uma mera notação amigável
- Pré-definida e definida pelo usuário
  - Tradicionalmente linguagens só tinham sobrecarga de operadores pré-definidos
- Excesso e mau uso de sobrecarga pode comprometer a legibilidade dos programas

Sobrecarga vs Polimorfismo

- Sobrecarga
  - Um mesmo identificador denotando diferentes entidades
    - (==), (+), (<), ...
- Polimorfismo
  - Uma única definição opera sobre uma **família de tipos**
    - `fst :: (t,r) -> t`
    - `head :: [t] -> t`
    - `length :: [t] -> Int`

Exercícios Recomendados

- Estabeleça a declaração genérica das funções e operadores:
  - `unzip`
  - `splitAt`
  - `reverse`
  - `replicate`
  - `concat`
  - `++`
  - `!!`
  - `:`

Generalização: Funções como argumento

- Reuso é uma meta principal na indústria de software
- Haskell permite definir funções gerais
  - Polimorfismo paramétrico
  - Funções como argumentos
- Funções como argumento permitem escrever funções que representam **padrões de computo**
  - Transformar todos os elementos de uma lista
  - Selecionar todos os elementos que atendem a uma propriedade
  - Combinar os elementos de uma lista usando um operador
- Chamamos estas funções de **combinadores**

Transformando Todos: map

```
roundAll :: [Float] -> [Int]
roundAll [ ] = [ ]
roundAll (y : ys) = round y : roundAll ys

maiusculo :: String -> String
maiusculo [ ] = [ ]
maiusculo (c : cs) = toUpper c : maiusculo cs

dodroLista :: [Int] -> [Int]
dobroLista [ ] = [ ]
dobroLista (x : xs) = 2*x : dobroLista xs
```

Característica comum: os elementos das listas estão sendo transformados pela aplicação de uma função.

Transformando Todos: map

- A ideia é usar a função que promove a transformação como argumento de uma função que faz a aplicação da transformação. Desta forma, várias funções podem ser usadas como argumento e uma mesma função aplicadora pode ser usada para transformar elementos de listas de várias maneiras.
- A função `map` é a função aplicadora e ela pode receber várias funções como argumento. Estas funções serão aplicadas por `map` aos elementos da lista.

```
map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (z:zs) = f z : map f zs
```

```
map :: (a -> b) -> [a] -> [b]
map f zs = [f z | z<-zs]
```

Transformando Todos: map

```
map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (z:zs) = f z : map f zs
```

```
roundAll :: [Float] -> [Int]
roundAll [ ] = [ ]
roundAll (y : ys) = round y : roundAll ys

roundAll :: [Float] -> [Int]
roundAll ys = map round ys
```

Transformando Todos: map

```
map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (z:zs) = f z : map f zs
```

```
maiusculo :: String -> String
maiusculo [ ] = [ ]
maiusculo (c : cs) = toUpper c : maiusculo cs

maiusculo :: String -> String
maiusculo cs = map toUpper cs
```

Transformando Todos: map

```
map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (z:zs) = f z : map f zs
```

```
dodroLista :: [Int] -> [Int]
dobroLista [ ] = [ ]
dobroLista (x : xs) = 2*x : dobroLista xs

dobroLista :: [Int] -> [Int]
dobroLista xs = map dobro xs
   where dobro x = 2*x
```

Combinando zip com map: zipWith

- A função zip permite agrupar duas listas numa só onde o elemento é um par
  - `zip :: [a] -> [b] -> [(a,b)]`
- Duas visões de `zipWith`
  - generalização de `map` para funções binárias, ou
  - generalização de `zip` onde o "critério de agrupamento" é dado como argumento
- Exemplo:
  - `zipWith (+) [1,2,3] [10,20,30,40] ↝ [1+10,2+20,3+30]`

Combinando zip com map: zipWith

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = [ ]
```

Qual o tipo de `zipWith`??

Combinando zip com map: zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = [ ]
```

```
ladoAlado :: Figura-> Figura
ladoAlado [] = []
ladoAlado (xs:xss) = (xs++xs): ladoAlado xss

ladoAlado2 :: Figura -> Figura -> Figura
ladoAlado2 xss xss = zipWith (++) xss xss
```

Combinando and/or com map: all/any

- A função `map` combinada com `and` permite identificar se todos os itens satisfazem uma dada propriedade `p`.
- A combinação de `map` com `or` identifica se algum item satisfaz uma dada propriedade `p`.

```
all :: (t -> Bool) -> [t] -> Bool
all p xs = and (map p xs)

any :: (t -> Bool) -> [t] -> Bool
any p xs = or (map p xs)
```

Combinando and/or com map: all/any

- A função `map` combinada com `and` permite identificar se todos os itens satisfazem uma dada propriedade.
- A combinação de `map` com `or` identifica se algum item satisfaz uma dada propriedade.

```
all :: (t -> Bool) -> [t] -> Bool
all p xs = and (map p xs)

any :: (t -> Bool) -> [t] -> Bool
any p xs = or (map p xs)
```

```
all odd [1,3,5,7] ↝ and [True,True,True,True] ↝ True
any even [1,3,5,7] ↝ or [False,False,False,False] ↝ False
```

Exercícios Recomendados

- Usando a função `map`, elabore:
  - Função para converter uma lista de caracteres numa lista dos códigos numéricos destes caracteres.
  - Função para converter uma lista de inteiros em uma lista dos quadrados destes inteiros.
  - Função para realizar o mesmo que refleteV no exemplo de figuras já visto.
  - Função para dado uma lista de pares retornar uma lista dos segundos elementos destes pares.
  - Função que computa o comprimento de uma lista sem usar a função `length` e usando `map` e `sum`.

Exercícios Recomendados

- Usando apenas a função `zipWith`, elabore:
  - Função para retornar uma lista contendo o dobro dos elementos de uma lista de inteiros.
  - Função para retornar uma lista contendo os quadrados dos elementos de uma lista de inteiros.
- Elabore uma função para verificar se dado uma string ela não contém dígitos.

Selecionando Alguns: filter

- Outro padrão de cômputo comum:
  - Escolher os elementos de uma lista que possuem uma dada propriedade
- Exemplos:
  - Escolher os números ímpares de uma lista de inteiros
  - Escolher dígitos de uma string
  - Escolher caracteres minúsculos de uma string

Selecionando Alguns: filter

```
impares :: [Int] -> [Int]
impares xs = [x | x <- xs, odd x]

digitos :: String -> String
digitos cs = [c| c <- cs, isDigit c]

minusculas :: String -> String
minusculas cs = [c| c<-cs, ehMinusculo c]
   where ehMinusculo c = ('a' <= c) && (c <= 'z')
```

Característica comum: os elementos das listas estão sendo selecionados pela aplicação de uma função. Eles atendem a uma dada propriedade.

Selecionando Alguns: filter

- Modelamos as propriedades como funções que retornam Bool
  - `odd :: Int -> Bool`
  - `isDigit :: Char-> Bool`
  - `ehMinusculo :: Char -> Bool`
- Um elemento `x` tem a propriedade `f` quando `f x == True`

```
filter :: (a -> Bool) -> [a] -> [a]
filter f zs = [z | z <- zs, f z]
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [ ] = [ ]
map f (z:zs)
    | f z          = z : filter f zs
    | otherwise = filter f zs
```

Selecionando Alguns: filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter f zs = [z | z <- zs, f z]
```

```
impares :: [Int] -> [Int]
impares xs = [x | x <- xs, odd x]

impares :: [Int] -> [Int]
impares xs = filter odd xs

digitos :: String -> String
digitos cs = [c| c <- cs, isDigit c]

digitos :: String -> String
digitos cs = filter isDigit cs
```

Selecionando Alguns: filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter f zs = [z | z <- zs, f z]
```

```
minusculas :: String -> String
minusculas cs = [c| c<-cs, ehMinusculo c]
   where ehMinusculo c = ('a' <= c) && (c <= 'z')

minusculas :: String -> String
minusculas cs = filter ehMinusculo cs
   where ehMinusculo c = ('a' <= c) && (c <= 'z')
```

Exercícios Recomendados

- Usando a função `filter`, elabore:
  - Função para selecionar os elementos positivos de uma lista de inteiros.
  - Função selecionar pares ordenados de uma lista de pares.
  - Função para selecionar pares na forma (a, a^2), de uma lista de pares.
  - Função para selecionar as listas inteiros ordenadas de uma lista de lista de inteiros
  - Função para dada uma string e um caracter `x` determinar a última posição de `x` na string, ou -1 se ele não ocorrer na string.

Exercícios Recomendados

- Usando a função `filter`, elabore:
  - Função para dada uma propriedade `p` e uma lista, remove da lista de entrada o primeiro elemento que não satisfaz `p`.
  - Função para dada uma propriedade `p` e uma lista, remove da lista de entrada o último elemento que não satisfaz `p`.
  - Função para selecionar os elementos que ocupem posições pares na lista de entrada.
  - Função para dada uma lista gerar um par onde o primeiro elemento do par contém a lista de elementos das posições pares e o segundo elemento do par a lista dos elementos das posições ímpares.
- Usando `filter` e `zipWith` elabore uma função para retornar uma lista contendo o dobro dos números ímpares de uma lista de inteiros.

Combinando elementos: foldr1 e foldr

- Outro padrão de cômputo comum:
  - Combinar todos os elementos de uma lista pela aplicação de uma operação ou função sobre estes elementos
- Exemplos:
  - Somar os números de uma lista de inteiros
  - Concatenar as listas de uma lista de listas
  - Calcular a conjunção dos elementos de uma lista de Booleanos
  - Calcular o máximo de uma lista de inteiros

Combinando elementos: foldr1 e foldr

```
sum [2,3,71] ↝ 2 + 3 + 71

concat [[1..5],[3,7,10],[16]] ↝ [1..5] ++ [3,7,10] ++ [16]

and [True, True, False] ↝ True && True && False

maior [2, 71, 40] ↝ 2 `max` (71 `max` 40)
```

Característica comum: os elementos das listas estão sendo combinados pela aplicação de uma função/operador, resultando num único resultado da função ou operação.

Combinando elementos: foldr1 e foldr

```
foldr1 g [e1,e2, ... ,ek]
 = e1 `g` (e2 `g` ... (... `g` ek) ... )
 = e1 `g` (foldr1 g [e2, ..., ek])
 = g e1 (foldr1 g [e2, ..., ek])
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 g [x] = x
foldr1 g (x:xs) = g x (foldr1 g xs)
```

Combinando elementos: foldr1 e foldr

```
foldr1 (+) [2,3,71] ↝ 2 + 3 + 71

foldr1 (++) [[1..5],[3,7,10],[16]] ↝ [1..5] ++ [3,7,10] ++ [16]

foldr1 (&&) [True, True, False] ↝ True && True && False

foldr1 max [2, 71, 40] ↝ 2 `max` (71 `max` 40)
```

Combinando elementos: foldr1 e foldr

- Mas `foldr1` não está definido para lista vazia. Precisamos estender a definição para lidar com este caso introduzindo um argumento extra, o valor retornado quando a lista for vazia.

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 g [x] = x
foldr1 g (x:xs) = g x (foldr1 g xs)
```

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

Combinando elementos: foldr1 e foldr

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

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs

and :: [Bool] -> Bool
and xs = foldr (&&) True xs
```

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

maior :: [Int] -> Int
maior [x] = x
maior (x:xs) = max x (maior xs)

maior :: [Int] -> Int
maior xs = foldr1 max xs
```

Observe que a função `maior` não está definida para lista vazia, assim pode-se usar `foldr1`.

Combinando elementos: foldr1 e foldr

- É possível generalizar a definição de `foldr` para contemplar o caso em que a função passada para aplicar o `foldr` não tem argumentos de mesmo tipo.

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

Combinando elementos: foldr1 e foldr

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

```
snoc :: a -> [a] -> [a]
snoc x xs = xs ++ [x]
```

```
reverse :: [t] -> [t]
reverse xs = foldr snoc [] xs
  where snoc x xs = xs ++ [x]
```

Combinando elementos: foldr1 e foldr

```
reverse :: [t] -> [t]
reverse xs = foldr snoc [] xs
  where snoc x xs = xs ++ [x]
```

```
reverse [4, 71, 40] ↝ foldr snoc [] [4, 71, 40]
                    ↝ 4 `snoc` (71 `snoc` (40 `snoc` []))
                    ↝ 4 `snoc` (71 `snoc` ([] ++ [40])
                    ↝ 4 `snoc` (71 `snoc` [40])
                    ↝ 4 `snoc` ([40] ++ [71])
                    ↝ 4 `snoc` [40,71])
                    ↝ [40,71] ++ [4] ↝ [40,71,4]
```

Combinando elementos: foldr1 e foldr

- Definições de funções recursivas sobre listas podem ser expressas sem usar a recursão usando `foldr`.
- Exemplo: `length` com `foldr`. Precisamos encontrar uma função `g` para aplicar o `foldr` que dê o efeito de length.

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

```
length :: [a]-> Int
length xs = foldr g 0 xs
  where g :: a -> Int -> Int
        g _ n = n+1
```

Combinando elementos: foldr1 e foldr

```
length :: [a]-> Int
length xs = foldr g 0 xs
  where g :: a -> Int -> Int
        g _ n = n+1
```

```
length [4, 71, 40] ⇝ foldr g 0 [4, 71, 40]
                   ⇝ 4 `g` (71 `g` (40 `g` 0))
                   ⇝ 4 `g` (71 `g` 1)
                   ⇝ 4 `g` 2
                   ⇝ 3
```

Combinando elementos: foldr1 e foldr

- Outro exemplo: `ordInsercao` com `foldr`.

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

```
ordInsercao :: [a]-> [a]
ordInsercao xs = foldr insOrd [] xs
  where insOrd :: Int -> [Int] -> [Int]
        insOrd y [] = [y]
        insOrd y (z:zs)
          | y <= z = y : z : zs
          | otherwise = z: insOrd y zs
```

Combinando elementos: foldr1 e foldr

```
ordInsercao :: [a]-> [a]
ordInsercao xs = foldr insOrd [] xs
  where insOrd :: Int -> [Int] -> [Int]
        insOrd y [] = [y]
        insOrd y (z:zs)
          | y <= z = y : z : zs
          | otherwise = z: insOrd y zs
```

```
ordInsercao [4, 71, 40] ↝ foldr insOrd [] [4, 71, 40]
                        ↝ 4 `insOrd` (71 `insOrd` (40 `insOrd` []))
                        ↝ 4 `insOrd` (71 `insOrd` [40])
                        ↝ 4 `insOrd` (71 `insOrd` 40:[])
                        ↝ 4 `insOrd` (40:71:[])
                        ↝ 4 `insOrd` (40:[71])
                        ↝ 4:40:[71] ↝ [4, 40, 71]
```

Exercícios Recomendados

- Usando a função `foldr` ou `foldr1`, elabore:
  - Função para efetuar o `or` de uma lista de Booleanos.
  - Função para realizar o produto de uma lista de reais.
  - Função para calcular o fatorial de um número inteiro.
  - Função para calcular o menor elemento de uma lista de inteiros.
  - Função para calcular o menor par de uma lista de pares de inteiros. Dados dois pares *(a,b)* e *(c,d)*. O par *(a,b)* é menor que o *(c,d)* se:

    $a < c$ ou

    $a = c$ e $b \leq d$.

Exercícios Recomendados

- Usando as funções `foldr`, `filter` e/ou `map`, elabore:
  - Função para efetuar a soma dos ímpares de uma lista de inteiros.
  - Função para realizar o produto dos quadrados de uma lista de inteiros.
  - Função para calcular o produto dos quadrados dos números maiores que 3 de uma lista de inteiros.
  - Função para calcular o menor elemento de uma lista de inteiros.
  - Função para calcular o menor par de uma lista de pares de inteiros. Dados dois pares *(a,b)* e *(c,d)*. O par *(a,b)* é menor que o *(c,d)* se:

    $a < c$ ou

    $a = c$ e $b \leq d$.

Exercícios Recomendados

- Defina uma função `filterFirst :: (a -> Bool) -> [a] -> [a]` tal que `filterFirst p xs` remove o primeiro elemento de `xs` que não satisfaz a propriedade `p`.
- Defina `filterLast :: (a -> Bool) -> [a] -> [a]` que remove a última ocorrência de um elemento de uma lista que não satisfaz a propriedade.
- Defina a função `switchMap` que aplica de forma alternada duas funções aos elementos de uma lista. Por exemplo

```
switchMap addOne addTen [1,2,3,4] ~> [2,12,4,14]
```

Generalizando divisão de listas

- Funções para quebrar listas já vistas
  - `take:: Int -> [a] -> [a]`
  - `drop :: Int -> [a] -> [a]`
  - `splitAt :: Int -> [a] -> ([a],[b])`
- Novas funções: coletam ou descartam elementos enquanto uma condição for verdadeira
  - `takeWhile:: (a -> Bool) -> [a] -> [a]`
  - `dropWhile :: (a -> Bool) -> [a] -> [a]`
  - `Ex: takeWhile isDigit "12bcd34gh" ↝ "12"`
    `dropWhile isDigit "12bcd34gh" ↝ "bcd34gh"`

Generalizando divisão de listas

```
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 = xs
```

Generalizando divisão de listas

- Redefinindo `insOrd` com `takeWhile` e `dropWhile`

```
insOrd :: Int -> [Int] -> [Int]
insOrd x xs = takeWhile menorx xs ++ [x] ++ dropWhile menorx xs
          where menorx y = y <= x
```

Exercícios Recomendados

- Dado um texto, do tipo `String`, defina as seguintes funções:
  - Coleta a primeira palavra do texto. Você deve supor que a palavra pode ser composta de qualquer caracter que não seja o branco, `\t` ou `\n`.

    ```
    primPalavra :: -> String -> String
    ```

  - Descarta a primeira palavra do texto.

    ```
    descPrimPal :: -> String -> String
    ```

  - Pula brancos, `\t` ou `\n` no início do texto.

    ```
    pulaDemarcadores :: -> String -> String
    ```

  - Coleta todas as palavras do texto, gerando uma lista de palavras.

    ```
    listaPalavras :: -> String -> [String]
    ```