Linguagens de Programação

Fabio Mascarenhas - 2015.2

http://www.dcc.ufrj.br/~fabiom/lp

Voltando a listas

- As funções que vimos para listas têm estruturas similares
- Elas codificam vários padrões:
 - Processar cada elemento da lista de certo jeito
 - Obter todos os elementos de uma lista que obedecem a algum critério
 - Combinar os elementos de uma lista usando alguma operação
- Uma linguagem funcional permite escrever funções genéricas para implementar esses padrões, usando funções de alta ordem

Aplicando uma função aos elementos

- Uma operação bem comum é processar cada elemento de uma lista, e agrupar os resultados em outra lista
- Por exemplo, para multiplicar os elementos de uma lista pelo mesmo fator, podemos escrever:

```
def mult(l: List[Double], fator: Double): List[Double] = 1 match {
  case Nil => Nil
  case h :: t => (h * fator) :: mult(t, fator)
}
```

map

 Podemos generalizar essa operação para qualquer lista e qualquer função, definindo a função map:

```
def map[T,U](1: List[T], f: T => U): List[U] # 1 match {
   case Nil => Nil
   case h :: t => f(h) :: map(t, f)
}
```

Usando map a definição da função mult fica mais concisa:

```
def mult(l: List[Double], fator: Double): List[Double] =
  map[Double](l, x => x * fator)
```

Filtragem

 Outra operação comum em listas é selecionar todos os elementos que satisfazem a determinada condição:

filter

• Esse padrão é generalizado pela função filter.

• E a função *filtraMaiores* fica com essa definição:

```
def filtraMaiores(l: List[Double], pivo: Double): List[Double] =
  filter[Double](l, x => x > pivo)
```

Redução

- Uma terceira operação bastante comum em listas é a redução
- Reduzir pegar os seus elementos e colocar alguma operação binária entre eles, e então avaliar o resultado

```
• Por exemplo, reduzir List(1,2,3,4,5) usando + é avaliar 1+2+3+4+5

def soma(l: List[Int]): Int = l match {
    case Nil => error("lista vazia")
    case h :: Nil => h
    case h :: t => h + soma(t)
}
```

Redução à direita

Generalizar a função soma nos dá uma redução à direita:

```
def reduceRight[T](l: List[T], op: (T, T) => T): T = 1 match {
  case Nil => error("lista vazia")
  case h :: Nil => h
  case h :: t => op(h, reduceRight(t, op))
}
```

• E *soma* pode ser expressa facilmente:

```
def soma(l: List[Int]): Int = reduceRight(l, (x, y) \Rightarrow x + y))
```

Mas a soma recursiva também tem uma versão com recursão final:

```
def soma(l: List[Int]): Int = {
    @tailrec
    def loop(l: List[Int], acum: Int): Int = l match {
        case Nil => acum
        case h :: t => loop(t, acum + h)
    }
    l match {
        case Nil => error("lista vazia")
        case h :: t => loop(t, h)
    }
}
```

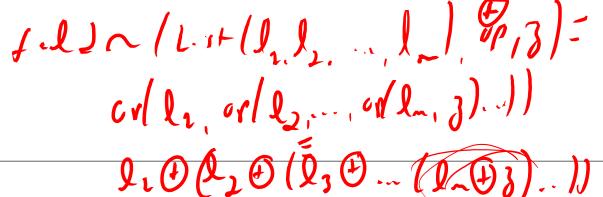
Redução à esquerda

 Generalizar a versão com recursão final de soma nos leva à redução à esquerda:

```
def reduceLeft[T](1: List[T], op: (T, T) => T): T = {
    @tailrec
    def loop(1: List[T], acum: T): T = 1 match {
        case Nil => acum
        case h :: t => loop(t, op(acum, h))
    }
    l match {
        case Nil => error("lista vazia")
        case h :: t => loop(t, h)
    }
}
```

 Se op é associativa, com a soma, podemos definir a redução de uma lista por op tanto à esquerda como à direita (à esquerda é mais eficiente!)

Folding



- As reduções já são operações bem gerais, mas ainda há mais umá generalização de reduceRight e reduceLeft
- Se, além da operação, fornecemos um elemento zero, podemos fazer reduções mesmo em listas vazias
- Além disso, o resultado da redução pode não ter o mesmo tipo que os elementos da lista!

• Um fold de uma lista / de elementos de tipo T usa um elemento zero z de tipo U e uma operação ou $(T, U) \Rightarrow U$ (para um fold à direita), ou $(U, T) \Rightarrow U$ (para um fold à esquerda)

foldRight e foldLeft

• As implementações de foldRight e foldLeft seguem da sua definição:

```
def foldRight[T,U](1: List[T], z: U, op: (T, U) => U): U = 1 match {
    case Nil => z
    case h :: t => op(h, foldRight(t, z, op))
}

@tailrec
def foldLeft[T,U](1: List[T], z: U, op: (U, T) => U): U = 1 match {
    case Nil => z
    case h :: t => foldLeft(t, op(z, h), op)
}
```

- Note como usamos o zero como um acumulador na definição de foldLeft!
- Não é difícil redefinir reduceRight e reduceLeft em função dos folds correspondentes, nem definir soma em função desses folds

Maps, filters e folds em Scala

- Esses padrões são tão comuns que naturalmente já existem implementações deles na biblioteca padrão de Scala
- Toda lista já tem funções map, filter (na verdade, vários tipos de filtro), reduce{Right, Left} e fold{Right, Left} definidas, chamadas com a sintaxe OO de Scala (usando.)

```
List(1,2,3,4,5).map(x => x * x)

List(1,2,3,4,5).foldLeft(1)((x, y) => x * y))

List(1,2,3,4).reduceRight((x, y) => x + y)
```