Programação Funcional
16ª Aula — Avaliação lazy e ordem superior revisitadas

Sandra Alves
DCC/FCUP

2019/20

Avaliação de expressões - I

Consideremos a seguinte função

```
quad ::  Int -> Int
quad x = x * x
```

A avaliação pode ser feita de dentro para fora (chamada-por-valor), por exemplo:

$$\begin{aligned} \texttt{quad}\ (3+4) &= \texttt{quad}\ 7 \\ &= 7 * 7 = 49 \end{aligned}$$

ou

$$\begin{aligned} \texttt{first}(\texttt{quad}\ 2, \texttt{quad}\ 4) &= \texttt{first}(2 * 2, \texttt{quad}\ 4) \\ &= \texttt{first}(4, \texttt{quad}\ 4) \\ &= \texttt{first}(4, 4 * 4) \\ &= \texttt{first}(4, 16) \\ &= 4 \end{aligned}$$

Avaliação de expressões - II

De fora para dentro (chamada-por-nome), por exemplo:

$$\begin{aligned} \texttt{quad}(3+4) &= (3+4)*(3+4) \\ &= 7*(3+4) \\ &= 7*7 \\ &= 49 \end{aligned}$$

ou

$$\begin{aligned} \texttt{first}(\texttt{quad}\ 2, \texttt{quad}\ 4) &= \texttt{quad}\ 2 \\ &= 2*2 \\ &= 4 \end{aligned}$$

De fora para dentro com partilha (chamada-por-necessidade), por exemplo:

$$\begin{aligned} \texttt{quad}(3+4) &= (3+4)*(3+4) \\ &= 7*7 \\ &= 49 \end{aligned}$$

Estratégias de avaliação - I

A ordem de avaliação das funções não altera o resultado (confluência)

A avaliação usando chamada-por-valor pode levar a reduções infinitas

Exemplo: `switch 0 (sum [1..]) 0`, com:

```
switch ::  Int -> Int -> Int -> Int
switch n e1 e2 | n > 0 = e1
               | otherwise = e2
```

A avaliação usando chamada-por-nome garante a terminação, caso o programa termine.

Estratégias de avaliação - I

A ordem de avaliação das funções tem impacto no número de reduções

A avaliação usando chamada-por-nome pode precisar de mais passos de redução

Argumentos duplicados, podem ser avaliados mais do que uma vez usando chamada-por-nome.

por nome

$$\begin{aligned} \texttt{quad}(3+4) &= (3+4)*(3+4) \\ &= 7*(3+4) \\ &= 7*7 \\ &= 49 \end{aligned}$$

por valor

$$\begin{aligned} \texttt{quad}\ (3+4) &= \texttt{quad}\ 7 \\ &= 7*7 \\ &= 49 \end{aligned}$$

Estratégia de avaliação lazy - I

A estratégia de avaliação usada no Haskell é a chamada-por-necessidade - *lazy evaluation*.

Os argumentos que não são usados, não são avaliados

Exemplo:

```
take 0 [1..10000]
```

Os argumentos podem ser avaliados parcialmente (só é avaliada a parte necessária para continuar a computação)

Exemplo:

```
take 10 [1..10000]
```

Estratégia de avaliação lazy - II

Um argumento duplicado é avaliado uma única vez (implementação baseada em grafos de partilha)

Exemplo, a avaliação de `quad (3+7)`:

$$\begin{aligned} \texttt{quad}(3+7) &= (3+7)*(3+7) \\ &= 10*10 \\ &= 100 \end{aligned}$$

Permite trabalhar com listas infinitas

Exemplo:

```
take 10 [1..]
```

Estratégia de avaliação lazy - III

Consideremos a expressão:

```
take 10 [1 | x <- [1..]]
```

Com avaliação lazy podemos separar os dados (a lista de 1's), da função que controla os dados (a função take).

Sem avaliação lazy teríamos que definir uma função para produzir uma lista de `n` elementos idênticos.

Permite maior modularização dos programas, o que é um objectivo importante em programação.

Nota: é necessário cuidado ao programar com listas infinitas:

`filter (<=5) [1..]` versus `takeWhile (<=5) [1..]`

Avaliação estrita (strict)

Uma desvantagem da avaliação lazy é a dificuldade de medir performance (tempo e espaço).

Por vezes pode ser útil forçar a avaliação de uma função de forma estrita.

Em Haskell podemos usar aplicação estrita `$!`.

Assim a expressão `f$!x` é avaliada normalmente, mas na aplicação de `f` a `x`, o argumento é forçosamente avaliado antes da aplicação da função.

Por exemplo, a aplicação `f x y` pode ser modificada para:

`(f$!x)y`	força a avaliação de `x`.
`(f x)$!y`	força a avaliação de `y`.
`(f$!x)$!y`	força a avaliação de `x` e `y`.

Ordem superior

Uma função é de ordem superior se:

- recebe como parâmetro uma função
- retorna uma função como resultado

Permite capturar padrões de programação.

As funções `map`, `foldr` e `filter` são funções de ordem superior em Haskell, que capturam padrões de programação.

Permite mais facilmente demonstrar propriedades de programas.

Exemplo : Dividir para conquistar

A técnica de dividir-para-conquistar é uma técnica fundamental no desenho de algoritmos (ex: quicksort)

Dado um determinado problema:

1. Dividimos o problema em problemas mais simples

2. Resolvemos cada um dos problemas separadamente (se cada um dos sub-problemas for suficientemente complicado podemos usar a mesma técnica recursivamente).

3. Combinamos as diferentes soluções dos sub-problemas, numa única solução

A função de ordem superior `divPConq` - I

Considerando um tipo `p` para os problemas e um tipo `s` para as soluções, a função `divPConq` tem os seguintes parâmetros:

- `ind :: p -> Bool`: que retorna `True` se o problema for indivisível e `False` caso contrário.
- `resolve :: p -> s`: que resolve uma instância indivisível de um problema.
- `divide :: p -> [p]`: que divide um problema numa lista de sub-problemas.
- `combina :: p -> [s] -> s`: dado o problema original e a lista de soluções para os sub-problemas, combina-as numa única solução.

A função de ordem superior `divPConq` - II

- Definimos a função dividir para conquistar da seguinte forma:

```
divPConq ::  (p -> Bool) -> (p -> s) -> (p ->[p])
-> (p -> [s] -> s) -> p -> s

divPConq ind resolve divide combina p
  = dc p
  where dc pi
    | ind pi = resolve pi
    | otherwise = combina pi (map dc (divide pi))
```

Dividir para conquistar: mergesort

Mergesort Divide a lista inicial em duas lista, recursivamente ordena as duas listas e depois junta-as numa lista ordenada.

Definimos o algoritmo de mergesort da seguinte forma:

```
msort xs = divPConq ind id divide combina xs
   where ind xs = length xs <= 1
         divide xs = let n = length xs 'div' 2
                     in [take n xs,drop n xs]
         combina _ [l1,l2] = merge l1 l2
```

Dividir para conquistar: quicksort

Quicksort: Depois de escolher o primeiro elemento da lista como pivot, separa o resto da lista em duas listas: os elementos menores ou iguais ao pivot e os elementos maiores que o pivot. A duas sublistas são ordenadas recursivamente e depois unidas com o pivot no meio.

Definimos o algoritmo de mergesort da seguinte forma:

```
qsort xs = divPConq ind id divide combina xs
   where ind xs = length xs == 0
         divide (x:xs) = [[y | y <- xs, y <= x]
                          [y | y <- xs, y > x]]
         combina (x:_) [l1,l2] = l1 ++ [x] ++ l2
```

Outras técnicas de implementação:

- Procura por geração e teste: problema das rainhas, etc
- Procura por prioridade
- Algoritmos “greedy”
- Ver: “Algorithms: A Functional Programming Approach”, Fethi Rabhi e Guy Lapalme