Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Avaliação Preguiçosa

(COMP0393)

Aula 17

dcomp
departamento de computação

Ordem de avaliação

- Ordem de cálculos não altera o resultado
- Haskell utiliza estratégia de avaliação preguiçosa
  - argumentos são avaliados por demanda
  - argumentos são avaliados uma única vez
  - somente são avaliadas as partes necessárias de um argumento estruturado
- É possível escrever estruturas potencialmente infinitas
- Tem implicações no projeto de programas

Avaliação por demanda

```
f x y = x + y
```

```
f (9-3) (f 34 3) ⇝ (9-3) + (f 34 3)
                 ⇝ 6 + (f 34 3)
                 ⇝ 6 + (34 + 3)
                 ⇝ 6 + 37
                 ⇝ 43
```

Aqui, os argumentos ainda não foram avaliados

Argumentos podem não ser avaliados

```
g x y = x+12
```

g (9-3) (g 34 3) ⇝ (9-3) + 12 ⇝ 6 + 12 ⇝ 18

```
switch :: Int → a → a → a
switch n x y
    | n > 0  = x
    | otherwise   = y
```

Aqui, ou `x` ou `y` serão avaliados mas não ambos

switch 2 (9-3) (34*8) ⇝ ?? 2>0 ⇝ ?? True ⇝ (9-3) ⇝ 6

Avaliação de ciclo curto é um caso particular de avaliação preguiçosa

```
(&&) :: Bool → Bool → Bool
True && x   = x
False && x  = False
```

Na segunda regra `x` não é avaliado

Argumentos são avaliados uma única vez

```
h x = x + x
```

```
h (9 -3) ⇝ (9-3) + (9-3)
         ⇝  6 + 6
         ⇝ 12
```

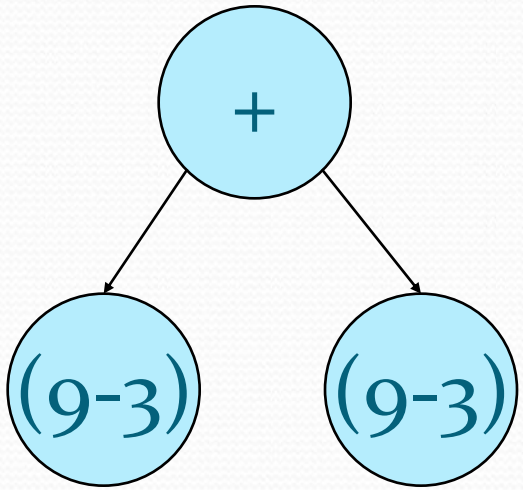
(9-3) será avaliado duas vezes? Não!

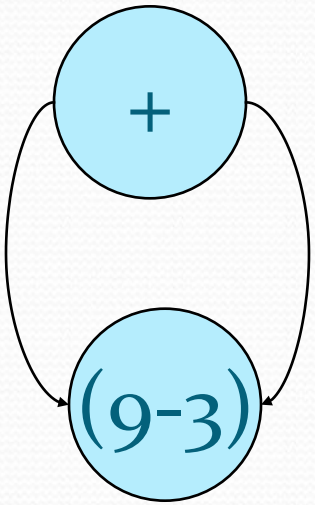
Num único passo

Após avaliado um argumento, o resultado é memorizado para no futuro usar o valor memorizado (*memoization*)

Implementação de *memoization*

A avaliação é feita sobre grafos, não em árvores





Avaliação sobre demanda em estruturas

```
sucH :: [Int] → Int
sucH (x:xs) = x + 1
sucH [] = 0
```

sucH [2+3, 4+5, 5+6] ⇝ sucH (2+3: [4+5, 5+6])
⇝ (2+3) + 1
⇝ 5 + 1
⇝ 6

Regras de cálculo

```
f p1 p2 ... pk
   | g1            = e1
   | g2            = e2
   ...
   | otherwise     = er
      where
         v1 a11 a12 ... = r1
         ...
f q1 q2 ... qk
```

- Para casamento, args. são avaliados o suficiente
- Equações tentadas em ordem
- Guardas são avaliadas em ordem
- Valores em **where** são calculados por demanda

Exemplos

```
f :: [Int] → [Int] → Int
f [ ] ys      = 0
f (x:xs) [ ] = 0
f (x:xs) (y:ys)    = x + y
```

```
   f [1..3][2..5]
↝ f (1:[2..3])[2..5]
↝   f (1:[2..3])(2:[3..5])
↝ 1+2
↝ 3
```

Exemplos

```
g :: Int → Int → Int → Int
g m n p
    | m>=n && m>=p = m
    | n>=m && n>= p      = n
    | otherwise          = p
```

```
g (2+3) (4-1) (3+9)
  ?? (2+3) >= (4-1) && (2+3) >= (3+9)
  ?? ↝ 5>=3 && 5>=(3+9)
  ?? ↝ True && 5>=(3+9)
  ?? ↝ 5>=(3+9)
  ?? ↝ 5>=12
  ?? ↝ False
  ?? 3>=5 && 3>=12
  ?? ↝ False && 3>=12
  ?? ↝ False
  ?? otherwise ↝ True
↝ 12
```

```
h :: Int → Int → Int
h m n
  | notNil xs = front xs
  | otherwise = n
  where
    xs = [m .. n]

front (x:y:zs)     = x+y
front [x]    = x

notNil [ ]   = False
notNil (_:_)= True
```

```
h 3 5
 ?? notNil xs
 ?? | where
 ?? | xs =[3..5]
 ?? | ↝ 3:[4..5]
 ?? ↝ notNil (3:[4..5])
 ?? ↝ True
↝ front xs
|     where
|     xs = 3:[4..5]
|       ↝ 3:4:[5]
↝ 3+4
↝ 7
```

Avaliação de outras construções

- Operadores embutidos
  - && e outros, normalmente (preguiçosa)
  - +, *, ... , necessita os dois argumentos
  - == varia
    - Em inteiros, necessita os dois argumentos
    - Em tipos compostos, avalia o necessário
- `if_then_else` avalia preguiçosamente
- `let` como `where`, por demanda
- Expressões lambda, similar a funções com nome
- Compreensões ...

Ordem de avaliação

- Avaliação normal
  - de fora para dentro

    ```
    f e1 (f e2 e3)
    ```

  - de esquerda para direita

    ```
    f e1 + f e2
    ```

- Avaliação preguiçosa = avaliação normal + *memoization*

Programação dirigida por dados

- Estruturas de dados intermediárias são construídas
- Queremos definir uma função para calcular as soma das potências de 4 desde 1 até n
  - Construímos a lista [1.. n]
  - Calculamos a potência de 4 de cada número [1,16,...,n^4]
  - Somamos a lista

```
sumPot4 :: Int → Int
sumPot4 n = sum (map (^4) [1..n])
```

```
sumPot4 :: Int → Int
sumPot4 n = sum (map (^4) [1..n])
```

```
sumPot4 n sum map
 ⇝ sum (map (^4) [1..n])
 ⇝ sum (map (^4) (1:[2..n]))
 ⇝ sum (1^4 : map (^4) [2..n])
 ⇝ 1^4 + sum ( map (^4) [2..n])
 ⇝ 1 + sum ( map (^4) [2..n])
 ⇝ ...
 ⇝ 1 + (16  + sum ( map (^4) [3..n]))
 ⇝ ...
 ⇝ 1 + (16 + (81 + ... + n^4))
```

Note que nenhuma lista intermediária é construída

Listas infinitas

- Avaliação preguiçosa permite descrever estruturas infinitas

```
ones :: [Int]
ones = 1 : ones

addFirstTwo :: [Int] → Int
addFirstTwo (x:y:zs) = x+y
```

```
addFirstTwo ones
 ⇝ addFirstTwo (1 : ones)
 ⇝ addFirstTwo (1 : 1: ones)
 ⇝ 1 + 1
 ⇝ 2
```

Exemplos

- Notação pré-definida para listas infinitas

```
[n .. ],   [n,m .. ]
```

```
pitagorasTri = [ (x,y,z) | z ← [2..], y ← [2 .. z-1],
                           x ← [2.. y-1], x^2 + y^2 == z^2]
```

```
pitagorasTri = [(3,4,5), (6,8,10), (5,12,13), (9,12,15),...]
```

Leitura Obrigatória

- Avaliação preguiçosa com compreensões

Exercícios Recomendados

- Exercícios 17.1 a 17.8 do Livro do Simon Thompson