Linguagens de Programação

Programação Funcional e Haskell
Avaliação Lazy

Thiago Alves

Introdução

Vamos avaliar com detalhes como as expressões são efetuadas em Haskell

Evita avaliação desnecessária

Chamada de avaliação lazy

Haskell é uma linguagem funcional lazy

Avaliação de Expressões

Expressões são efetuadas ou reduzidas através da aplicação sucessiva de definições até mais nenhuma simplificação for possível.

```
square n = n * n
```

```
 square(3 + 4)
=
 square 7
=
 7 * 7
=
 49
```

Temos outra redução possível:

```
 square(3 + 4)
=
 (3 + 4) * (3 + 4)
=
 7 * (3 + 4)
=
 7 * 7
=
 49
```

Aplicamos square antes de realizar a adição, mas o resultado final é o mesmo.

Em Haskell, duas formas diferentes de avaliar uma mesma expressão sempre retorna o mesmo resultado final.

As duas formas de avaliar a expressão devem terminar o processo

Essa propriedade não vale para a maioria das linguagens de programação imperativas

Assumindo que n tem valor inicial 0:

```
 n + (n := 1)
= {aplicando n}
 0 + (n := 1)
=
 0 + 1
=
 1
```

Essa propriedade não vale para a maioria das linguagens de programação imperativas

Assumindo que n tem valor inicial 0:

```
 n + (n := 1)
= {aplicando o :=}
 n + 1
=
 1 + 1
=
 2
```

Estratégias de Avaliação

Redução mais interna

A redução sempre é feita na subexpressão que não possui outra subexpressão reduzível

Se tiver mais que uma mais interna, escolhemos a que fica mais à esquerda

Redução mais Interna

```
mult(x,y) = x * y
```

```
 mult(1+2,2+3)
=
 mult(3,2+3)
=
 mult(3,5)
=
 3 * 5
=
15
```

Redução mais Interna

- Usar essa estratégia garante que os argumentos de uma função são efetuados antes da aplicação da função
  - Passagem por valor

Estratégias de Avaliação

Redução mais externa

A redução sempre é realizada na subexpressão que não está contida em nenhuma outra expressão reduzível

Se tiver mais que uma mais externa, a que fica mais à esquerda é escolhida

Redução mais Externa

```
mult(x,y) = x * y
```

```
 mult(1+2,2+3)
=
 (1+2) * (2+3)
=
 3 * (2+3)
=
 3 * 5
=
15
```

Redução mais Externa

- Permite que a função seja aplicada antes de efetuar os argumentos
  - Passagem por nome

- Algumas funções pré-definidas como * e + necessitam que os argumentos já tenham sido avaliados

Terminação

```
loop = tail loop
```

Usando redução mais interna:

```
 fst(1, loop)
=
 fst(1, tail loop)
=
 fst(1, tail (tail loop))
=
 ...
```

Não termina!

Terminação

```
loop = tail loop
```

Usando redução mais externa:

```
 fst(1, loop)
=
 1
```

Número de Reduções

```
square n = n * n
```

Usando redução mais interna:

```
 square(3 + 4)
=
 square 7
=
 7 * 7
=
 49
```

Número de Reduções

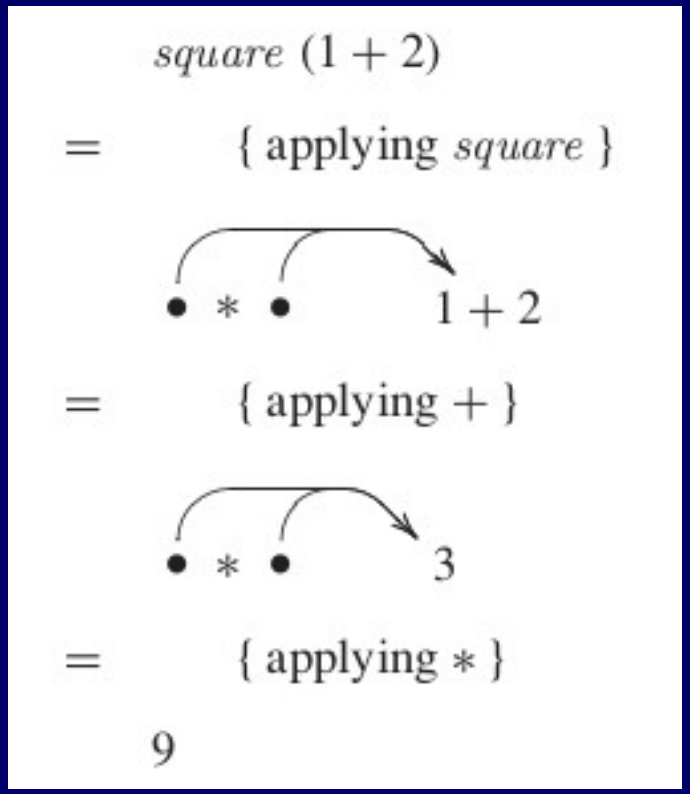
Usando redução mais externa:

```
  square(3 + 4)
=
  (3 + 4) * (3 + 4)
=
  7 * (3 + 4)
=
  7 * 7
=
  49
```

Redução mais externa pode precisar de mais passos que a redução mais interna.

Número de Reduções

O problema pode ser resolvido usando ponteiros para indicar o compartilhamento de expressões:



Número de Reduções

O uso de redução mais externa com o compartilhamento é chamado de avaliação lazy.

Nunca necessita de mais passos que a redução mais interna.

Listas Infinitas

Usar avaliação lazy permite programar com listas infinitas

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

```
ones = 1 : ones
       = 1 : 1 : ones
       = 1 : 1 : 1 : ones
     ...
```

Listas Infinitas

Vamos analisar com redução mais interna:

```
head ones = head (1 : ones)
          = head (1 : 1 : ones)
          = head (1 : 1 : 1 : ones)
          = ...
```

Não termina!

Listas Infinitas

Vamos analisar com avaliação lazy:

```
head ones = head (1 : ones)
          = 1
```

Apenas o primeiro valor na lista infinita é realmente produzido

Listas Infinitas

Usando avaliação lazy, as expressões são efetuadas apenas o necessário para produzir o resultado final

```
nats = [0..]
pares = [0,2..]
ints n = n : ints (n+1)
```

```
take 10 pares
take 5 (map (3*) nats)
head (tail (ints 5))
```

Listas Infinitas

Cuidados devem ser tomados para evitar a não terminação:

```
map (*3) nats
filter (<= 5) [1..]
```

No lugar do `filter` podemos usar:

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

Vamos criar a função que preenche uma string com espaços para deixá-la com n caracteres:

```
preencher :: Int → String → String
```

Exemplo:

```
> preencher 10 "Haskell"
"Haskell "
> preencher 10 "Haskell B. Curry"
"Haskell B."
```

Vamos criar a função que preenche uma string com espaços para deixá-la com n caracteres:

```
spaces = ' ' : spaces
preencher n xs = take n (xs ++ spaces)
```