Avaliação Lazy

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```
module Main where
```

Objetivos

- Entender a diferença entre as estratégias lazy e estrita para avaliação.
- Utilizar estruturas de dados infinitas.
- Entender falhas comuns relacionadas à avaliação lazy.
  - Utilizar `seq` para forçar a avaliação.

Uma expressão simples

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

$> square (1 + 2)
= -- magicamente temos que...
9
```

- Como o valor final foi obtido?

Avaliação estrita

- Maioria das linguagens utiliza avaliação estrita.
  1. Avalia argumentos de uma função completamente.
  2. Avalia a chamada de função.

```
square (1 + 2)
= -- avalia o argumento
square 3
= -- executa o corpo da função
3 * 3
= 9
```

Avaliação lazy

- Argumentos são substituídos diretamente no corpo da função.

```
square (1 + 2)
= -- substituindo no corpo da função
(1 + 2) * (1 + 2)
= -- avaliando (1 + 2) para continuar
3 * (1 + 2)
=
3 * 3
=
9
```

Avaliação lazy

- Mas, faz sentido usar avaliação lazy?
  - Para uma função como `square` não...
  - Na verdade, a avaliação lazy é mais custosa para `square`!
- Porém, sempre a avaliação lazy é pior?

Avaliação lazy

```
const x y = y -- ignora x

-- estrita
const (1 + 2) 5
=
const 3 5
=
5
```

```
-- lazy
const (1 + 2) 5
=
5
```

Sharing

```
square (1 + 2)
=
(1 + 2) * (1 + 2)
```

- Porque refazer o cálculo da expressão `(1 + 2)`?
  - Podemos compartilhar o resultado de avaliação, evitando o retrabalho.

Avaliação lazy

- Haskell utiliza avaliação lazy.
  - Expressões não são executadas até que sejam necessárias para o resultado final.
  - Expressões duplicadas são “compartilhadas”.

Avaliação lazy

- Seguindo essa estratégia, a avaliação lazy nunca requer mais passos que a avaliação estrita.
- Cada expressão aguardando avaliação é armazenada como uma estrutura de dados conhecida como **thunk**.

Avaliação lazy

- É possível obter resultados diferentes usando diferentes estratégias de avaliação?

Avaliação lazy

- É possível obter resultados diferentes usando diferentes estratégias de avaliação?

Avaliação lazy

- O teorema de Church-Rosser garante que o resultado será o mesmo independente da estratégia utilizada para programas que sempre terminam.
- Mas...
  - A performance pode ser diferente (consumo de memória).

Terminação

```
loop :: a -> a
loop x = loop x
```

- Esse programa é aceito pelo compilador.
- Mas a chamada `loop True` não termina.

Terminação

```
-- estrita
const (loop True) 5
```

```
-- lazy
const (loop True) 5
```

Terminação

```
-- estrita
const (loop True) 5
=
const (loop True) 5
=
```

```
-- lazy
const (loop True) 5
=
5
```

Terminação

```
-- estrita
const (loop True) 5
=
const (loop True) 5
=
const (loop True) 5
= ...
```

```
-- lazy
const (loop True) 5
=
5
```

Terminação

- A estratégia lazy considera que mais programas terminam quando comparada à avaliação estrita.

Estruturas de controle

```
if_ :: Bool -> a -> a -> a
if_ True  x _ = x
if_ False _ y = y
```

- Em linguagens estritas, `if_` deve avaliar ambos os branches.
  - Porém, isso não é o ideal.

Estruturas de controle

- Em linguagens lazy, somente o branch a ser executado.

Por essa razão:

- Em linguagens estritas, o `if` deve ser `built-in`.
- Em linguagens lazy, você pode construir suas próprias estruturas de controle.

Curto-circuito

```
(&&) :: Bool -> Bool -> Bool
True  && b = b
False && _ = False
```

Curto-circuito

- Em linguagens estritas, `x && y` deve avaliar ambos os argumentos
  - Por isso, Java / C / C++ / C# possuem conjunção built-in com suporte a curto circuito.

Curto-circuito

- Em Haskell, `x && y` avalia o segundo argumento somente se `x = True`.
  - Por isso, `False && (loop True)` termina.

Estruturas de dados

- Uma aplicação interessante da avaliação lazy é permitir estruturas de dados "infinitas".
- Exemplo:

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

Estruturas de dados

- `ones` é uma lista infinita, mas o programa que a usa não entra em loop desde que avalie um prefixo finito.

```
take 2 ones =
```

Estruturas de dados

- `ones` é uma lista infinita, mas o programa que a usa não entra em loop desde que avalie um prefixo finito.

```
take 2 ones =
take 2 (1 : ones) =
```

Estruturas de dados

- `ones` é uma lista infinita, mas o programa que a usa não entra em loop desde que avalie um prefixo finito.

```
take 2 ones =
take 2 (1 : ones) =
1 : take 1 ones   =
```

Estruturas de dados

- `ones` é uma lista infinita, mas o programa que a usa não entra em loop desde que avalie um prefixo finito.

```
take 2 ones =
take 2 (1 : ones) =
1 : take 1 ones   =
1 : take 1 (1 : ones) =
```

Estruturas de dados

- `ones` é uma lista infinita, mas o programa que a usa não entra em loop desde que avalie um prefixo finito.

```
take 2 ones =
take 2 (1 : ones) =
1 : take 1 ones   =
1 : take 1 (1 : ones) =
1 : 1 : take 0 ones =
```

Estruturas de dados

- ones é uma lista infinita, mas o programa que a usa não entra em loop desde que avalie um prefixo finito.

```
take 2 ones =
take 2 (1 : ones) =
1 : take 1 ones   =
1 : take 1 (1 : ones) =
1 : 1 : take 0 ones =
1 : 1 : [] = [1,1]
```

Estruturas de dados

- Construindo uma lista de todos os números naturais:

```
nats :: [Integer]
nats = 0 : map (+ 1) nats

take 2 nats =
```

Estruturas de dados

- Construindo uma lista de todos os números naturais:

```
nats :: [Integer]
nats = 0 : map (+ 1) nats

take 2 nats =
take 2 (0 : map (+1) nats) =
```

Estruturas de dados

- Construindo uma lista de todos os números naturais:

```
nats :: [Integer]
nats = 0 : map (+ 1) nats

take 2 nats =
take 2 (0 : map (+1) nats) =
0 : take 1 (map (+1) nats) =
```

Estruturas de dados

- Construindo uma lista de todos os números naturais:

```
nats :: [Integer]
nats = 0 : map (+ 1) nats

take 2 nats =
take 2 (0 : map (+1) nats) =
0 : take 1 (map (+1) nats) =
0 : take 1 (map (+1) (0 : map (+1) nats)) =
```

Estruturas de dados

- Construindo uma lista de todos os números naturais:

```
nats :: [Integer]
nats = 0 : map (+ 1) nats

take 2 nats =
take 2 (0 : map (+1) nats) =
0 : take 1 (map (+1) nats) =
0 : take 1 (map (+1) (0 : map (+1) nats)) =
0 : take 1 (1 : map (+1) (map (+1) nats)) =
```

Estruturas de dados

- Construindo uma lista de todos os números naturais:

```
nats :: [Integer]
nats = 0 : map (+ 1) nats

take 2 nats =
take 2 (0 : map (+1) nats) =
0 : take 1 (map (+1) nats) =
0 : take 1 (map (+1) (0 : map (+1) nats)) =
0 : take 1 (1 : map (+1) (map (+1) nats)) =
0 : 1 : take 0 (map (+1) (map (+1) nats)) =
```

Estruturas de dados

- Construindo uma lista de todos os números naturais:

```
nats :: [Integer]
nats = 0 : map (+ 1) nats

take 2 nats =
take 2 (0 : map (+1) nats) =
0 : take 1 (map (+1) nats) =
0 : take 1 (map (+1) (0 : map (+1) nats)) =
0 : take 1 (1 : map (+1) (map (+1) nats)) =
0 : 1 : take 0 (map (+1) (map (+1) nats)) =
0 : 1 : [] = [0,1]
```

Avaliação lazy

- Como Haskell determina quando deve avaliar ou não argumentos?
  - Por padrão, nada é avaliado.
  - Quando temos casamento de padrão, avalia-se até determinar qual equação deve ser executada.

Avaliação lazy

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

- Se `n = 0`, não há necessidade de avaliar a lista.
- Caso contrário, devemos distinguir se a lista é vazia ou não.

Avaliação lazy

- Uma expressão está na **weak head normal form** se
  - Se é um construtor com expressões não avaliadas.
- Uma função anônima.
- Uma função built-in parcialmente aplicada.

`True` ou `Just (1 + 2)`

De maneira precisa, argumentos são avaliados até a WHNF para decidir qual equação deve ser executada.

Exemplo

- Sequência de Fibonacci:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

- Como construir uma lista com toda a sequência de Fibonacci?

Exemplo

```haskell
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Exemplo

```
   0 : 1 : ... -- fibs
+  1 : ...     -- tail fibs
-------------------------------
   1 : ...     -- zipWith (+) ...
```

Exemplo

```
   0 : 1 : ... -- fibs
+  1 : 1 : ... -- tail fibs
-------------------------------
   1 : 2 ...   -- zipWith (+) ...
```

Exemplo

```
   0 : 1 : 1 : ... -- fibs
+  1 : 1 : 2 : ... -- tail fibs
-------------------------------
   1 : 2 : 3 : ... -- zipWith (+) ...
```

Exemplo

- O crivo de Eratóstenes é um algoritmo para enumerar todos os números primos.
  - Desenvolvido por Eratóstenes na antiguidade.

1. Liste todos os números iniciando com 2.
2. Seja p o primeiro número desta lista.
3. Remova todos os múltiplos de p da lista.
4. Retorne ao passo 2.

Exemplo

1. Liste todos os números iniciando com 2.

```
primes :: [Integer]
primes = sieve [2 ..]
```

Exemplo

2. Seja p o primeiro número da lista

```
primes :: [Integer]
primes = sieve [2 ..]

sieve :: [Integer] -> [Integer]
sieve (p : ns) = ...
```

Exemplo

3. Remova todos os números de p da lista.

```
primes :: [Integer]
primes = sieve [2 ..]

sieve :: [Integer] -> [Integer]
sieve (p : ns) = p : [ n | n <- ns, n `mod` p /= 0 ]
```

Exemplo

4. Retorne ao passo 2 e escolha o próximo número da lista.
   - Isto é, chame recursivamente!

```
primes :: [Integer]
primes = sieve [2 ..]

sieve :: [Integer] -> [Integer]
sieve (p : ns) = p : sieve [ n | n <- ns
                               , n `mod` p /= 0 ]
```

Funções lazy e estritas

- O que há de diferente com essas funções?

```
loop (2 + 3)
= -- definição de loop
loop (2 + 3)
= -- e lá vamos nós...

const 3 (loop 2)
= -- definição de const
3
```

Funções lazy e estritas

- Dizemos que uma função é **estrita** em um argumento se esta não terminada dado um valor que não termina a execução.
  - A função $(+)$ é estrita em ambos os argumentos.
  - A função const é estrita no primeiro.

Funções lazy e estritas

- Representamos não terminação por $\perp$ ou `undefined`.
  - Dizemos que f é estrita se $f \perp = \perp$

Exemplos

Qual o resultado das seguintes expressões?

1. `(\x -> x) True`
2. `(\x -> x) undefined`
3. `(\x -> 0) undefined`
4. `(\x -> undefined) 0`
5. `(\ x f -> f x) undefined`
6. `undefined undefined`
7. `length (map undefined [1,2])`

Exemplos

Qual o resultado das seguintes expressões?

1. `(\x -> x) True == True`
2. `(\x -> x) undefined == undefined`
3. `(\x -> 0) undefined == 0`
4. `(\x -> undefined) 0 == undefined`
5. `(\ x f -> f x) undefined == \f -> f undefined`
6. `undefined undefined == undefined`
7. `length (map undefined [1,2]) == 2`

Garbage collection

- Expressões (thunks) são gerenciados pelo runtime de Haskell.
  - Criados quando precisamos de um valor.
  - Não são excluídos assim que o valor final é obtido.

Garbage collection

- Haskell usa um Garbage Collector.
  - Periodicamente, GC libera memória não utilizada.
  - Vantagem: Não há necessidade de preocupar com alocação.
  - Desvantagem: GC consome tempo...
- Maioria das linguagens atuais usa GC!

Forçando a avaliação

- A função

```
seq :: a -> b -> b
```

força a computação do primeiro parâmetro antes de continuar a execução do segundo.

Forçando a avaliação

- Usando seq podemos definir um operador de avaliação estrita:

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Finalizando

- Avaliação lazy é guiada por pattern matching.
- Se não usada com cuidado leva a ineficiência
  - Mais thunks, mais consumo de memória
  - Mais memória usada, mais GC e isso demanda tempo...

Exercícios

Considere o tipo de dados Stream, de lista infinitas:

```
data Stream a
   = a :> Stream a
     deriving (Eq, Ord, Show)
```

1. Crie a função:

```
streamToList :: Stream a -> [a]
streamToList = undefined
```

que converte um Stream em uma lista.

Exercícios

2. Crie a função

```
streamMap :: (a -> b) -> Stream a -> Stream b
streamMap f _ = undefined
```

que aplica uma função a todos os elementos de um Stream