Recursão, Listas e Funções de Ordem Superior

Programação Funcional

Prof. Maycon Amaro

Recursividade

- Visto em Estrutura de Dados I
- Vamos *revisitar*

Recursão é quando uma função chama a si mesma.

```
int fatorial(int x) {
  if (x == 0)
    return 1;
  else
    return x * fatorial(x - 1);
}
```

Chamadas recursivas são colocadas na pilha de execução. Dependendo do número de chamadas a serem realizas, a pilha pode “estourar”.

Recursão de cauda

Também conhecido como **tail recursion**, é quando a chamada recursiva é o único retorno ou comando da função.

Geralmente, conseguimos transformar uma recursão em recursão de cauda transformando alguns valores em parâmetros adicionais, que servem como acumuladores.

```
int _fatorial(int x, int acc) {
  if (x == 0)
    return acc;
  else
    return _fatorial(x - 1, x * acc);
}

int fatorial(int x) {
  return _fatorial(x, 1);
```

Não há razão para manter chamadas anteriores na pilha. O resultado da última chamada é o resultado da chamada original.

Bons compiladores identificam a recursão de cauda e realizam essa otimização.

Complexidade

Funções recursivas são sucintas e elegantes, mas podem ter uma complexidade exponencial.

```
int fibonacci(int x) {
  if (x == 0 || x == 1)
    return x;
  else
    return fibonacci(x-1) + fibonacci(x-2);
}
```

Usando recursão de cauda, podemos melhorar a complexidade da fibonacci para linear.

```
int _fibonacci(int x, int a, int b) {
  if (x == 0)
    return a;
  else
    return _fibonacci(x-1, b, a+b);
}

int fibonacci(int x) {
  return _fibonacci(x, 0, 1);
}
```

Recursão na avaliação *lazy*

Em linguagens funcionais com avaliação estrita, a recursão de cauda pode ter um desempenho bem melhor que uma forma mais "geral" de recursão.

Em Haskell, que tem avaliação preguiçosa, o cenário é mais complicado.

Um valor só é realmente computado quando outro que precisa dele é avaliado.

O GHC é capaz de realizar excelentes otimizações de código nas funções recursivas.

Por causa da complexidade, a versão com recursão de cauda de fibonacci abaixo é também mais eficiente em Haskell.

```haskell
fibonacci :: Int -> Int
fibonacci x = fib x 0 1
  where
    fib 0 a b = a
    fib y a b = fib (y-1) b (a+b)
```

No entanto, a versão com recursão de cauda do fatorial não necessariamente traz vantagens.

```haskell
fact :: Int -> Int
fact x = fact' x 1
  where
    fact' 0 acc = acc
    fact' x acc = fact' (x - 1) (x * acc)
```

Programação declarativa

> *Diga o que você quer, deixe o compilador decidir como será feito.*

Devemos nos atentar à complexidade das funções que escrevemos. Mas não se preocupe tanto com aspectos que não afetam a complexidade.

Utilizar as funções do prelúdio sempre que possível é uma boa dica.

Listas

Listas

Como já vimos no λ-cálculo, listas são estruturas essencialmente recursivas.

- `[]` é uma lista, a lista vazia.
- Se `xs` é uma lista, então um elemento `x` adicionado a `xs`, escrito como `x:xs`, é uma lista.

O tipo das listas é representado com colchetes. Assim, o tipo `[a]` significa lista de elementos do tipo `a`.

```
exemplo :: [Int]
exemplo = [2, 1, 6]

exemplo :: [Int]
exemplo = 2 : 1 : 6 : []
```

Podemos realizar casamento de padrão sobre listas.

```
ehVazia :: [Int] -> Bool
ehVazia [] = True
ehVazia  _ = False

contarElementos :: [Int] -> Int
contarElementos [] = 0
contarElementos (x:xs) = 1 + contarElementos xs
```

Essas funções estão disponíveis no prelúdio como `null` e `length`.

Com um ajuste na função `contarElementos` podemos obter uma `somarElementos`, que está disponível no prelúdio como `sum`:

```
somarElementos :: [Int] -> Int
somarElementos [] = 0
somarElementos (x:xs) = x + somarElementos xs
```

Funções de Ordem Superior

Funções de Ordem Superior

No λ-cálculo, qualquer termo pode ser passado como argumento para uma abstração, inclusive outra abstração.

$$(\lambda x.x\ c_1)\ (\lambda y.y)$$

Em linguagens de programação, quando uma função espera uma função como parâmetro, dizemos que ela é uma **função de ordem superior**.

```
id1 :: Int -> Int
id1 x = x

id2 :: (Int -> Int) -> Int
id2 f = f 1
```

Padrões de Recursão

Vamos ver alguns exemplos de recursão que parecem ter uma estrutura muito parecida.

Padrão 1

Aplicar uma função em cada elemento de uma lista, construindo uma nova lista.

```
dobro :: [Int] -> [Int]
dobro [] = []
dobro (x:xs) = x * 2 : dobro xs

ehPar :: [Int] -> [Bool]
ehPar [] = []
ehPar (x:xs) = even x : ehPar xs
```

Generalizando com uma função de ordem superior.

```
aplicarEmTudo :: (Int -> a) -> [Int] -> [a]
aplicarEmTudo _ [] = []
aplicarEmTudo f (x:xs) = f x : aplicarEmTudo f xs

dobro :: [Int] -> [Int]
dobro xs = aplicarEmTudo (*2) xs

ehPar :: [Int] -> [Bool]
ehPar xs = aplicarEmTudo even xs
```

`aplicarEmTudo` está no prelúdio como `map` e funciona com listas de qualquer tipo.

Padrão 2

Selecionar elementos que satisfazem uma condição.

```
soPares :: [Int] -> [Int]
soPares [] = []
soPares (x:xs)
  | even x = x : soPares xs
  | otherwise = soPares xs

soPositivos :: [Int] -> [Int]
soPositivos [] = []
soPositivos (x:xs)
  | x > 0 = x : soPositivos xs
  | otherwise = soPositivos xs
```

Generalizando com uma função de ordem superior.

```haskell
selecionar :: (Int -> Bool) -> [Int] -> [Int]
selecionar _ [] = []
selecionar f (x:xs)
  | f x = x : selecionar xs
  | otherwise = selecionar xs

soPares :: [Int] -> [Int]
soPares xs = selecionar even xs

soPositivos :: [Int] -> [Int]
soPositivos xs = selecionar (>0) xs
```

`selecionar` está no prelúdio como `filter`.

Padrão 3

Acumular aplicações de uma função sobre os elementos da lista.

```
somarElementos :: [Int] -> Int
somarElementos [] = 0
somarElementos (x:xs) = (+) x (somarElementos xs)

multElementos :: [Int] -> Int
multElementos [] = 1
multElementos (x:xs) = (*) x (multElementos xs)
```

Generalizando com uma função de ordem superior.

```
acumular :: (Int -> Int -> Int) -> Int -> [Int] -> Int
acumular _ v [] = v
acumular f v (x:xs) = f x (acumular f xs v)

somarElementos :: [Int] -> Int
somarElementos xs = acumular (+) 0 xs

multElementos :: [Int] -> Int
multElementos xs = acumular (*) 1 xs
```

`acumular` está no prelúdio como `foldr`, que acumula aplicando da direita para a esquerda.

Sem valor inicial?

Use `foldr1` ao invés: `foldr1 (+) xs`

Em outras linguagens

Python

```
xs = map(lambda x : x + 1, [1, 2, 3, 4])
print(list(xs))
#[2, 3, 4, 5]

ys = filter(lambda x : x % 2 == 0, [1, 2, 3, 4])
print(list(ys))
#[2, 4]

zs = reduce(lambda x, y : x + y, [1, 2, 3, 4])
print(zs)
#10
```

As funções `map` e `filter` em Python funcionam sobre iteradores, o que as tornam *lazy*. O uso de `list()` força a avaliação.

JavaScript

```
[1, 2, 3, 4].map(x => x + 1)
//[2, 3, 4, 5]

[1, 2, 3, 4].filter(x => x % 2 === 0)
//[2, 4]

[1, 2, 3, 4].reduce((x, y) => x + y)
//10
```

Observação: a avaliação é estrita. É possível obter o comportamento *lazy* com uso de algumas técnicas.

Rust

```rust
let xs = vec![1, 2, 3, 4];

let ys = xs.iter().map(|x| x + 1).collect();

println!("{:?}", ys);
//[2, 3, 4, 5];
```

Em Rust, as funções de ordem superior também funcionam sobre iteradores. `collect()` força a avaliação.

C/C++

Ainda prefere fazer isso em C?

```
int v[] = {1, 2, 3, 4};
int soma = 0;
for(int i = 0; i < n; i++)
  soma += v[i];
}
printf("%d", soma);
```

C++ já virou a página... (não é *lazy*)

```
std::vector<int> v = {1, 2, 3, 4};
int soma =
  std::reduce(v.cbegin(), v.cend(), 0, std::plus);
std::cout << soma;
```

Conclusão

- Recursão pode ser simplificada com uso de funções de ordem superior.
- Outras linguagens incorporaram essa funcionalidade, incluindo o comportamento *lazy*.
- Listas são uma ótima estrutura para usar recursividade. Os exercícios serão sobre elas.
- Outras funções de ordem superior estão disponíveis na biblioteca padrão de Haskell e das outras linguagens. Dê um Google!