Resumo Conceitos 2020

Lourenço Henrique Moinheiro Martins Sborz Bogo - 11208005

17 de janeiro de 2021

Sumário

1 Funcional **2**
1.1 Introdução . . . 2
1.2 Propriedades Principais . . . 2
1.3 Lambdas e Closures . . . 2
1.4 Currying . . . 3
1.5 Continuações . . . 3
1.6 Recursão . . . 3
1.7 Polimorfismo . . . 4
1.8 Propósito . . . 4

2 Lazy Evaluation **4**
2.1 Introdução . . . 4
2.2 Suspensões . . . 5
2.3 Vantages e Desvantagens . . . 5

3 POO **6**
3.1 Introdução . . . 6
3.2 Objetos . . . 6
3.3 Classes . . . 6
3.4 Implementações de Herança . . . 7
3.5 Polimorfismo . . . 7
3.6 Diferentes tipos de OO . . . 7
3.7 Implementação . . . 7

1 Funcional

1.1 Introdução

Programação funcional foi o primeiro tópico que vimos na matéria.

Trata-se de um paradigma onde as funções são valores, ou seja, podemos passá-las como argumentos e atribuí-las a variáveis. Além disso, o paradigma possui outras propriedades relevantes que irei abordar em seguida.

1.2 Propriedades Principais

Uma parte extremamente importante de linguagens funcionais é que, em geral, os procedimentos (funções) não têm efeitos colaterais: uma função apenas recebe os seus argumentos, executa algum tipo de operação com eles e no final retorna um novo valor. O procedimento não altera variáveis de outro escopo, inclusive os seus argumentos, que permanecerão inalterados no ambiente do escopo no qual o procedimento foi chamado.

As funções podem ser argumentos de outras funções, o que nos conduz a um ponto crucial do paradigma funcional: as funções de ordem superior. Essas são funções cujos parâmetros são outras funções e, quando combinadas com certas propriedades do paradigma, nos permitem atingir um nível de abstração muito interessante nas linguagens. Com base nesse conceito podemos criar alguns dos procedimentos mais importantes e conhecidos da programação funcional, como o `map`, o `filter` e o `reduce`.

1.3 Lambdas e Closures

Sendo as nossas funções valores, não é necessário que tenham nomes e, por essa razão, são definidas de maneira anônima. Denominamos funções anônimas como lambdas. Se quisermos dar-lhes um nome, podemos simplesmente atribuí-las a alguma variável, já que elas são valores comuns iguais a `1` ou `"foo"`.

Outro conceito é o fechamento (closure), que é um par `(lambda, ambiente)`. Quando criamos uma nova função usando `lambda`, montamos um fechamendo com a lambda previamente definida e o ambiente atual é estendido pela variável dessa lambda. Isso é feito por dois motivos principais:

1. No momento de fazer a aplicação da função definida, vamos ao seu ambiente e alteramos o valor da variável para o valor passado como argumento na aplicação.

2. Como todo ambiente de fechamento é uma extensão do ambiente no qual esse fechamento foi criado, as funções mais internas poderão acessar os argumentos das funções externas.

Vale relembrar aqui a diferença entre escopo léxico e escopo dinâmico. O que usamos em nossos trabalhos foi o escopo léxico, onde uma variável terá o valor que lhe foi dado localmente, pois expandimos os ambientes da maneira que foi explicada acima. Já no escopo dinâmico, todo identificador terá associada uma pilha global de valores. Quando criamos uma variável cujo nome é esse identificador, inserimos o valor atribuído à variável na pilha do identificador. Ao avaliarmos certa variável, o valor resultante será o que estiver no topo da pilha. O desempilhamento é feito seguindo os escopos.

1.4 Currying

Outro ponto interessante abordado foi o conceito de `curry` que, de forma sintética e simplista, significa transformar funções de múltiplos parâmetros em cadeias de funções com apenas um parâmetro. Exemplificando:

```
;; Função definida sem currying
(define somaDois
  (lambda (x y)
    (+ x y)))

;; Função definida com currying
(define somaDoisCurry
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

A primeira função só pode ser chamada quando passamos dois argumentos e ela somará os dois. A segunda função recebe um argumento, devolve um fechamento, que também recebe um argumento, e só então soma os dois. Isso é útil, pois torna o código mais reutilizável e sustentável. Por exemplo, se quisermos fazer um procedimento que recebe um argumento e acrescenta 1 a esse argumento, poderíamos reutilizar `somaDoisCurry`, e definir a nova função mais facilmente:

```
(define somaUm
  (somaDoisCurry 1)) ;; Retorna uma lambda de um argumento que soma 1 a esse argumento
```

1.5 Continuações

Aprendemos também sobre continuações. Ao invés de termos uma função que recebe e retorna valores, adcionamos a essa função outro parâmetro que será um procedimento. O argumento desse procedimento será o valor que a função estava retornando anteriormente. Ou seja:

- Fazíamos $f(x) \rightarrow y$
- Passamos a fazer $f(x, g) \rightarrow g(y)$. y aqui é o que a função retornava no caso anterior. Nesse exemplo, g é denominada a continuação de f.

A grande utilidade disso é que nos permite controlar melhor o fluxo do programa e tratar mais facilmente situações não usuais, como por exemplo, sair de recursões sem termos que voltar por todas as chamadas.

1.6 Recursão

Como nas linguagens funcionais não podemos ter mutações, não temos loops, portanto, fazemos ações repetidas com o auxílio de recursão. Porém, recursão pode acabar sendo ineficiente em alguns casos, por exemplo, quando queremos calcular o fatorial de um número. O jeito trivial de fazer isso seria:

```
(define fatorial
  (lambda (x)
    (if (equal? 0 x) 1 (* x (fatorial (- x 1))))))
```

Isso é ineficiente, pois podemos acabar enchendo a pilha com chamadas recursivas para fatorial, caso nossa entrada seja um inteiro muito grande.

Podemos resolver isso usando recursão de cauda! Faríamos da seguinte maneira:

```
(define fatorialaux
  (lambda (x acc)
    (if (equal x 0) acc (fatorial (- x 1) (* acc x)))))

(define fatorial
  (lambda (x)
    (fatorialaux x 1)))
```

Como podemos ver, agora a nossa função `fatorialaux` não faz nada após as chamadas recursivas, ou seja, a função é uma recursão de cauda. Isso é útil no quesito eficiência, pois os interpretadores/compiladores da grande maioria das linguagens otimizam recursões de cauda para virarem loops, então esse código roda sem desperdiçar a pilha de execução.

1.7 Polimorfismo

Em programação funcional temos também o conceito de polimorfismo, que é criar uma interface única para funções de vários tipos. Isso é útil pois podemos fazer funções como o mapc que pode ser aplicada a uma lista de qualquer tipo (inclusive em linguagens tipadas como Haskell). Isso dá uma quantidade imensa de expressividade para linguagens funcionais (fica mais evidente em linguagens fortemente tipadas).

1.8 Propósito

Todas essas propriedades fazem linguagens funcionais parecerem menos úteis que linguages procedurais, mas isso não é verdade. Programação funcional funciona muito melhor que programação procedural/orientada a objetos para resolver problemas que têm uma modelagem matemática precisa, pois basicamente o que fazemos nesse paradigma é encadear funções sem efeitos colaterais, assim como fazemos na matemática.

2 Lazy Evaluation

2.1 Introdução

O segundo tópico abordado na matéria foi "Lazy Evaluation"(Avaliação por Demanda).

Avaliação por demanda é uma estratégia de avaliação cujo princípio básico é: apenas calcular o valor de uma expressão quando esse valor for necessário. É o contrário do que vínhamos aprendendo até agora, que era avaliação ansiosa, cujo princípio é avaliar uma expressão na primeira vez que ela for encontrada.

Para ilustrar a diferença entre esses dois métodos de avaliação, segue um exemplo simples:

```
(cons (+ 1 2) '())
```

O código acima monta uma lista cujo primeiro elemento é a aplicação da função + nos elementos 1 e 2. Se estamos em uma linguagem onde temos avaliação ansiosa, acontecerá o seguinte: ao encontrarmos a operação `(+ 1 2)`, iremos avaliá-la e teremos como resultado 3. Nosso código então irá produzir uma lista com apenas 1 elemento, que será o número 3.

Fica claro que não usamos o valor 3 para nada, ele não foi necessário para nenhuma operação. Para montarmos a lista, não precisávamos saber que ao avaliarmos aquela expressão teríamos como resultado 3. Para isso que serve a avaliação por demanda. No código acima, se estivermos em uma linguagem que implementa esse tipo de avaliação, não iremos calcular o valor da soma, iremos criar uma lista de 1 elemento, cujo valor é uma **SUSPENSÃO**, que quando avaliada irá nos retornar o valor 3.

2.2 Suspensões

Suspensões são uma estrutura semelhante a um fechamento sem argumentos, elas guardam uma expressão e um ambiente. Quando o valor da suspensão for necessário, iremos interpretar a expressão dessa suspensão, com o ambiente contido nela.

Por eficiência, sempre que avaliamos pela primeira vez uma suspensão específica, substituímos no ambiente global o seu valor pelo valor retornado dessa avaliação. Assim, na próxima vez que precisarmos do valor dessa suspensão, poderemos utilizá-la sem ter que recalcular o seu valor.

Para todo esse sistema funcionar, necessitamos de um novo tipo de funções que serão chamadas **Funções Estritas**. Essas funções irão avaliar seus argumentos imediatamente, ou seja, caso recebam uma suspensão como argumento, elas irão expandir essa suspensão. Exemplificando:

```
(if (equal? (+ 1 2) 3) (alguma_coisa) (outra_coisa))
```

Nesse caso, para podermos continuar o programa, é necessário que avaliemos o valor da condição do if imediatamente. Ou seja, a condição do if é estrita, significando que ela sempre irá avaliar as suspensões dadas.

Alguns outros exemplos de funções estritas são:

- Operações aritméticas, já que precisamos saber em quais valores estamos aplicando essa operação (não faz sentido somar duas suspensões)
- Car e cdr, já que quando queremos um elemento de uma lista, queremos o elemento em si e não uma suspensão

2.3 Vantages e Desvantagens

Primeiro, a grande desvantagem da avaliação por demanda é a seguinte: já que uma suspensão guarda também um ambiente, dependendo do jeito que implementarmos esse sistema, pode ser que o custo de memória fique imprevisível, podendo ser muito alto, muito baixo ou até mesmo negativo, já que vamos evitar a avaliação de partes não usadas de estruturas recursivas . Além disso, funções com efeitos colaterais podem quebrar as coisas, já que podemos alterar suspensões antes de termos usado seu valor para o que queríamos.

Agora, as vantagens principais desse método de avaliação são:

- Aumento na performance da linguagem, já que iremos evitar avaliações desnecessárias
- Podemos ter estruturas de dados infinitas, pois só iremos calcular os elementos necessários dessa estrutura. Exemplo em haskell:

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
      -- Código retirado de wiki.haskell.org/The_Fibonacci_sequence
      -- Nesse exemplo montamos uma lista que contém TODOS os números da
      -- sequência de Fibonacci. O único problema com isso é que se pedirmos algo que
      -- exige a lista toda (como o tamanho da lista), o programa irá quebrar.
```

3 POO

3.1 Introdução

O terceiro e último grande tópico abordado foi o paradigma de programação conhecido como Orientação a Objetos. O paradigma foi inventado com o propósito de podermos abstrair nossos dados e esconder alguns detalhes de suas representações. Os códigos desse paradigma tendem a ficar mais modularizados e intuitivos.

Como já dito antes, o paradigma tem como objetivo principal a abstração dos dados, ou seja, podemos montar `classes` para representar mais abstratamente o dado que temos. Isso é muito útil, pois nos permite organizar nossos dados com mais facilidade e de maneira mais intuitiva.

3.2 Objetos

Tudo (ou quase tudo, dependendo da linguagem) nesse paradigma é representado por objetos. Um objeto é formado por um conjunto de dados e um conjunto de procedimentos (métodos), que nos permite alterar esses dados e produzir valores.

Isso nos conduz ao primeiro ponto principal do paradigma que é o encapsulamento. Só podemos acessar e alterar os dados de um certo objeto através da sua interface de funções (os métodos), nos dando duas vantagens principais:

- Abstração, já que alteramos o estado do objeto através de métodos que podem ser extremamente complexos
- Segurança, já que como o único jeito de alterar o estado do objeto é através dos métodos, se os métodos garantidamente sempre produzirem novos estados consistentes, não conseguiremos quebrar o programa.

Outra coisa essencial do paradigma é que, como já mencionado, podemos (e devemos) alterar os estados dos objetos, ou seja, temos efeitos colaterais, diferente do paradigma funcional, onde tínhamos idealmente apenas funções puras.

3.3 Classes

Uma classe é como uma fábrica de objetos de uma certa categoria. Todos os objetos criados a partir de uma classe terão um escopo interno com mesmos nomes e os mesmos métodos. Classes seguem uma estrutura hierárquica, ou seja, podemos definir uma classe B que é 'filha' de uma certa classe A. Nesse caso, dizemos que B está **herdando** de A, e temos as seguintes propriedades:

- B terá todas as variáveis de instância que A
- B terá os mesmos métodos que A
- B pode definir novas variáveis e métodos (incremento)
- B pode **REDEFINIR** os métodos de A (Redefinição)

Ao invés de herdar, podemos também delegar o trabalho para um objeto de outra classe, ou seja, podemos ter uma certa classe D, que contém um objeto da classe C. Desse modo, podemos usar os métodos definidos em C usando o objeto dentro de D.

Delegar é quase sempre melhor que herdar, a grande exceção pra isso é quando queremos usar a propriedade de redefinição da herança. Nesses casos, herdar é vantajoso.

3.4 Implementações de Herança

Em Java e Smalltalk, fazemos uma busca dinâmica pelos métodos, ou seja, quando queremos enviar uma mensagem (chamar um método), o que fazemos é procurar pelo nome desse método na classe e nas superclasses do objeto para o qual a mensagem está sendo enviada.

Já em C++, fazemos a execução direta do código, ou seja, cada objeto inclui ponteiros para cada um dos métodos de suas classes. Isso torna C++ a linguagem orientada a objetos mais rápida que existe e que pode existir, porém tira um pouco de expressividade.

3.5 Polimorfismo

Herança nos permite o uso de uma propriedade chamada polimorfismo. Suponhamos que temos uma classe Animal. Dessa classe, herdamos uma outra classe Cachorro e uma outra classe Gato. Cachorro e Gato podem ter implemetações diferentes para os métodos da classe animal (usando redefinição), porém uma função que recebe um Animal irá funcionar igualmente bem em objetos da classe Cachorro e objetos da classe Gato, já que os dois herdam da mesma classe Animal.

Isso é polimorfismo e é uma das propriedades da programação orientada a objeto que dá mais expressividade para o paradigma.

3.6 Diferentes tipos de OO

Em C++, tipos são basicamente a mesma coisa que classe. Precisamos redefinir métodos explicitamente usando a palavra chave `virtual` e não temos o conceito de interface, o mais próximos que podemos usar são classes abstratas puras.

Já em Smalltalk, tipos são apenas conjuntos de métodos e como a verificação dos tipos é feita dinamicamente, temos polimorfismo natural.

Em Java, tipos é uma classe mais uma interface implementada por essa classe. O polimorfismo é garantido pelo pelo uso das interfaces.

3.7 Implementação

Para implementarmos POO, temos que ter algumas coisas em mente:

- Um objeto sempre armazena o conteúdo das suas variáveis de instância
- O código dos métodos é compartilhado entre objetos de mesma classe, assim ocupamos menos espaço
- Todos os métodos têm um parâmetro implícito que é o próprio objeto que está chamado esse método. Assim os métodos podem alterar o estado desse objeto.