MAC0316 - Conceitos de Linguagens de Programação
Texto Avaliativo

Arthur Font Gouveia – 12036152

1. Linguagens funcionais

1.1. Paradigma funcional

As linguagens de programação funcionais são regidas por um paradigma funcional, no qual definem-se funções matématicas ao invés de escrever um programa ou uma lista de instruções como na programação orientada a objetos ou imperativa. A partir de dados de entrada, estas funções matemáticas analisam e calculam várias operações, obtendo uma saída.

1.2. Efeitos colaterais

As funções não possuem efeitos colaterais, isto é, as operações realizadas durante a execução não alteram o valor dos dados de entrada. Além disso, no paradigma funcional não existem variáveis, mas sim constantes. Visto isso, o código em programação funcional tende a ser mais breve, objetivo e de fácil manutenção.

Observa-se que no para manipular uma lista, é necessária a criação de uma nova lista que compartilha partes da lista anterior devido a inexistência de efeitos colaterais, que não permite a alteração de qualquer elemento de uma lista.

1.3. Recursão em cauda

Em uma recursão comum, a cada chamada recursiva, a pilha que armazena a posição onde foi feita a chamada aumenta. Logo, um número muito grande de chamadas pode levar ao estouro da pilha de memória, isso ocorre quando um programa tenta alocar mais memória do que tem disponível.

A fim de solucionar este problema, surgiu a recursão de cauda, que é uma estrutura de controle recursiva eficiente pois utiliza a cauda da recursão como sua última chamada, tornando desnecessário o armazenameneto do ponteiro com o endereço de retorno. Deste modo, o retorno da função que realizou a chamada recursiva é o mesmo retorno da função chamada ao princípio, o que garante que nenhum tratamento é feito antes de retornar o valor da última chamada.

1.4. LISP e o conceito de ambiente

LISP é uma família de linguagens funcionais interpretadas criada por John McCarthy em 1958. A sintaxe é simples e sua principal estrutura de controle é a recursão, já que não existem estruturas de repetição predefindas.

Em LISP é possível adicionar variáveis e ser capaz de manter valores para estas variáveis. Para isso, utilliza-se uma lista de associações para associar um valor ao símbolo de uma variável, esta lista de associações é chamada de ambiente ou e*nvironment*.

O LISP também dispõe de um coletor de lixo que tem como objetivo desalocar a memória que não está sendo mais utilizada, assim como em Java. Este coletor é acionado de maneira implícita e automática ao detectar as células que não estão sendo utilizadas, como por exemplo as células que não tenha mais utilidade após a manipulação de uma lista.

1.5. Ambiente

Em nossa interpretador desenvolvido no *racket*, o ambiente corresponde a uma lista de *Bindings* que relaciona o símbolo da variável a um valor numérico. A lista pode ser definida da seguinte forma:

```
( define-type Binding
    [ bind ( name : symbol ) ( val : number ) ])
```

Logo, na função **interp** (que corresponde ao nosso interpretador), chamamos a função **lookup**, que busca de maneira recursiva o identificador passado como argumento na lista de associações ou ambiente, a fim de verificar se esse identificador está ou não na lista, evitando possíveis erros. Uma vez encontrado o identificador, é realizado a troca a partir das associações.

1.6. Fechamento

Um fechamento ou *closure* é uma função interna que possui acesso a variáveis independentes da função externa, permitindo assim que as associações se propagaguem apenas para as funções que sejam internas ao seu contexto. Para isso, inicialmente o *enviroment* é vazio, e a cada vez ocorre uma aplicação, o *enviroment* é estendido com a associação da variável ao seu valor.

O fechamento nada mais é que um pacote que contém a função e o *enviroment* correspondente e pode ser ilustrado a seguir da seguinte forma:

```
<< lambda-expr, ambiente>>
```

Logo, para calcular o valor de uma expressão simbólica basta calcular o valor da no ambiente, pois algumas variáveis da expressão podem não serem parâmetros. Portanto o ambiente é responsável por atribuir valores a estas variáveis, chamadas de variáveis livres.

1.7. Açúcar sintático

Açúcar sintático é uma sintaxe que tem como objetivo de simplificar as construções da linguagem, tornando-a mais fácil de ser lida e codificada. O açúcar sintático serve tanto para manter a linguagem central pequena, como para estender a linguagem como um todo, o que possibilita a remoção do açúcar sem nenhum efeito sobre a linguagem.

A importância do açúcar sintático se dá pela facilidade de reutilização e manipulação das funções, já que as funções podem ser expressas através de uma sintaxe muito mais simples e robusta.

2. Avaliação por demanda

2.1. Vantagens e problemas programação funcional

Como visto anteriorente, a programação funcional utiliza fechamentos para obter maior poder de expressão e flexibilidade. Também utiliza açúcar sintático para estender a linguagem como um todo, o que torna o código do paradigma fucncional mais reutilizável e modular.

Entretanto, o paradigma funcional é ineficiente em algumas operações, como por exemplo na geração de listas, onde somente é possível gerar a lista inteira, e não somente as células necessárias para a execução da função.

2.2. Avaliação por demanda

Para solucionar este problema, utiliza-se a técnica da *lazy evaluation* ou avaliação por demanda. Segundo esta técnica, deve-se calcular o valor da expressão somente quando seja necessário por meio do adiamento das operações, o que aumenta o desempenho do evitando cálculos desnecessários e permite o controle de estruturas de dados infinitas. No caso das listas, o seu tamanho é irrelevante, já que calcula-se apenas as células que serão acessadas.

2.3. Suspensão

O adiamento das operações é realizada através da supensão ou *thunk*, para isso deve ser criada uma estrutura especial ao invés de calcular a expressão encontrada. Esta estrutura é semelhante a um fechamento sem argumentos e está ilustrada a seguir:

<| expressão, ambiente |>

Segue abaixo um exemplo de uma suspensão para a função **interval**, que utiliza a avalição por demanda para percorrer o intervalo de 1 a 100:

<| (interval 1 n), {n → 100} |>

A avaliação das expressões na avaliação por demanda funciona da seguinte maneira. Primeiramente, calcula-se a primeira expressão e contrói-se as suspensões para todas as expressões.

Posteriormente, deve-se analisar qual é o operador. Para o operador condicional e o estrito, deve-se avaliar as expressões normalmente. Para o operador “cons” deve-e criar uma célula e inserir as suspensões no primeiro e no segundo elemento desta célula. Para os fechamentos, deve-se calcular a expressão no ambiente aonde o valor será a suspensão referente. Por fim, para o operador “car” ou “cdr”, se o elemento for uma suspensão, deve-se calcula-lá e substituída pelo valor calculado, em seguida deve-se retornar o valor.

2.4. Crítica sobre a avalição por demandas

A avaliação por demanda torna o código difícil de ser depurado, pois o fluxo é determinado pela necessidade do argumento conforme a execução da função. Portanto, os resultados podem ser afetados pela sequência da exucação, o que torna a avaliação por demanda passível de efeitos colaterais. Por outro lado, a avaliação por demanda permite a manipulação de listas infinitas sem a necessidade de estabelecer um limite. Aliás, a operação de criação de uma suspensão é bastante rápida, o que torna o tempo de execução potencialmente menor nos casos em que hajam expressões que não serão necessárias em uma função. A avaliação por demanda também contribui para o uso eficiente da memória.

2.5. Scheme e avaliação por demanda

A linguagem funcional Scheme viola o princípio da avaliação por demanda, já que atua sobre o princípio de avaliação gulosa, no qual calcula-se o valor dos argumentos de uma função antes de que ela seja chamada pelo usuário.

3. Programação orientada a objetos

3.1. Introdução

A Programação Orientada a Objetos é um dos paradigmas de programação de software baseado na composição e interação entre diversas unidades chamadas de

objetos. Este paradigma é centrado na definição de conjunto de dados semelhantes, ao contrário de funções (como no paradigma funcional), tendo os efeitos colaterais como parte integrante.

Este modelo surgiu com o intuito de aproximar o manuseio das estruturas de um programa ao nossa visão do mundo real. Para isso, este paradigma utiliza dois conceitos chaves, classes e objetos.

3.2. Classes e objetos

As classes são abstrações dos dados que possuem características ou comportamentos semelhantes. Esta abstração tem como objetivo esconder a representação dos dados, permitindo ao usuário a manipulação abstrata do progama e não da expressão concreta a ser realizada.

Os objetos são definidos por um agrupamento de funções e são basicamente materialização das classes, sendo que cada materialização é chamada de instância. Portanto, o objeto é um conjunto com um estado interno que pode ser alterado conforme o programa se executa. O acesso aos dados é somente possível atraveś de funções através do encapsulamento, que será explicado na próxima seção.

3.3. Mensagens e métodos

Os métodos são descrições do comportamento e dos procedimentos que podem ser aplicado aos dados internos, isto é, a um objeto de uma classe. Uma mensagem é uma invocação de um dos métodos de um objeto.

3.4. Encapsulamento

O encapsulamento é uma técnica que adiciona segurança ao programa, já que esconde as propriedades do usuário. Para isso, adiciona-se uma camada de segurança ao objeto, evitando-se assim acesso direto às propriedades do objeto. O encapsulamento garante a abstração e a manutenção da consistência dos dados, proporcionando assim a modularidade necessária aos programas.

3.5. Herança e o conceito de tipo

A herança é uma característica da Programação Orientada a Objetos que proporciona a reutilização de granularidade fina do código e estruturas de dados, facilitando a redefinição de funções.

Através da herança é possível estabelecar uma hierarquia de classes, onde os objetos de uma classe filha compartilham das características da classe mãe como os métodos e as variáveis de instância e podem ser especilalizadas atravrés do incremento de novos métodos e as variáveis de instância. É importante notar que a herança se contrapõe a delegação do paradigma funcional.

Tipo é um conceito abstrato que indica as operações que pode ser executadas e podem ou não estar relacionada a classes. Em C++ um tipo é uma classe, em Java um tipo pode ser uma classe ou uma interface, já em Smalltalk um tipo é um conjunto de métodos.

A implementação da herança possui relação direta com o tipo. Em C++ o objeto possui endereço dos métodos de suas classes, o que torna o envio de mensagens em tempo de execução constante, porém proporciona menor flexibidade. Java consegue unir flexibilidade da resolução dinâmica às vantagens da tipagem estática através do polimorfismo realizado por classes que implementam interfaces, que garante que objetos entendam as mensagens enviadas a eles.

3.6. Polimorfismo

O polimorfismo é uma característica da Programação Orientada a Objetos que permite que referências de classes mais abstratas representem o corportamento de classes mais concretas, esse conceito está diretamente ligado à herança.

3.7. Smalltalk e a implementação da POO em nosso interpretador

Smalltalk é uma linguagem pura de POO (Programação Orientada a Objetos) dinâmicamente tipada pois tudo são objetos e o controle de fluxo é realizado somente através de mensagens e objetos do tipo *closure*.

Em nosso interpretador, utilizamos a resolução dinâmica dos métodos em uma versão híbrida do Smalltalk já que inteiros e booleanos são primitivos e existem operações que não são envio de mensagens. A linguagem inicial foi estendida para conter as principais características da POO como a implementação de classes, objetos e métodos (incluindo a chamada realizada através de mensagens), além da implementação da herança e do polimorfismo.