Category Theory Arrows e Monads em Java

Mozart L. Siqueira
Ciências da Computação
Centro Universitário La Salle - Unilasalle
Email: mozarts@unilasalle.edu.br

Pablo M. Parada
Ciências da Computação
Centro Universitário La Salle - Unilasalle
Email: pablo.paradabol@gmail.com

***Resumo*—<escrever>**
***Index Terms*—<escrever>**

I. INTRODUÇÃO

<escrever>

II. SOBRE O PARADIGMA FUNCIONAL

Influenciado principalmente pelo desenvolvimento do lambda calculus [1], compondo o grupo da programação declarativa, o paradigma funcional utiliza-se da idéia de expressar computações através de funções combinadas em expressões. Neste, funções determinam o que deverá ser computado, ao invés de como sera computado [2]. Programas são construídos através da composição, tal que funções triviais (ou building blocks) são combinadas dando origem a novas funções que descrevem computações mais complexas.

Building blocks não devem fazer uso de variáveis que dependam de estado, isso significa que a computação deve ser pura e sem efeitos indesejados (ou side-effects). Também destaca-se o princípio de imutabilidade, onde o valor é de uma variável é determinado em sua criação, não permitindo novas atribuições posteriormente.

É possível afirmar que ao expressar um programa em uma linguagem funcional, obtem-se uma maneira concisa de solucionar problemas, dado que este constitui-se de operações e objetos atômicos e regras gerais para sua composição [3]. Estas qualidades são apreciadas nos tempos atuais, onde há necessidade de tratar os problemas oriundos do não-determinismo. Assim, o paradigma funcional mostra-se capaz, inclusive de influenciar outras linguagens como Java [4].

A. Lambda Expressions e Anonymous Inner Classes

Ao fornecer funções de primeira classe (também lambda expressions ou closures), a linguagem Java habilita a substituição de annonymous inner classes (AIC) de forma transparente. Contudo, apesar destas expressarem o mesmo comportamento a nível de código, ambas funcionalidades possuem diferentes implementações sob a Máquina Virtual Java (JVM). Por exemplo, na odernação de inteiros podemos utilizar uma AIC em conjunto do método sort da classe Arrays tal que

```
Integer[] integers = new Integer[]{5, 4, 3, 2, 1};

Arrays.sort(integers, new Comparator<Integer>() {
    public int compare(Integer a, Integer b) {
        return a.compareTo(b);
    }
});
```

Listing 1. Sort - Anonymous Inner Class

obtem-se como retorno do método, um array com elementos ordenados. Entretanto, o mesmo método pode ser simplificado por uma lambda expression através de

```
Arrays.sort(integers, (a, b) -> a.compareTo(b));
```

Listing 2. Lambda Expressions

que descreve a mesma expressão sem os encargos impostos por AIC.

Conforme afirmou-se anteriormente, AIC e closures de fato diferem sob a JVM. AIC são compiladas, ou seja, geram novos arquivos contendo declarações de classes. Além do mais, ao utilizar a palavra reservada **this** referencia-se a própria instância anônima. Como representam instâncias de uma classe, estas devem ser carregadas pelo class loader e seus construtores invocados pela máquina virtual. Ambas etapas consomem memória, tanto heap para alocação de objetos quanto permgem, utilizada para guardar metadados, definições de classes e métodos [5].

Diferentemente de AIC, lambdas postergam a estratégia de compilação para em tempo de execução, utilizando a instrução invokedynamic [6]. Funções são traduzidas para métodos estáticos vinculados ao arquivo da classe correspondente a sua declaração, eliminando o consumo de memória. Agora, ao referir-se a **this**, a classe que delimita a lamda expression é acessada, ao contrário de AIC que acessa sua própria instância. Por fim, closures fornecem formas mais expressivas de representar comportamentos.

III. CATEGORY THEORY E SUAS APLICAÇÕES

Afim de demonstrar as relações entre estruras e sistemas matemáticos [7], a Category Theory (CT) estabelece uma linguagem formal capaz de encontrar aplicabilidade em várias áreas da ciência [8]. Na computação seu emprego fornece abstrações capazes de demonstrar as estruturas de programas e como dá-se sua composição. Com base nisto, Haskell foi capaz de resolver um dos problemas do The Awkward Squad [9], onde fundamentalmente laziness e side-effects eram incompatíveis até o surgimento de monads [10] na linguagem. A partir de então, diversas outras estruturas como categories, functors, arrows foram implementadas em Haskell.

A. *Category – Objetos e Morfismos*

Uma category C consiste em uma coleção de objetos e uma coleção de relações entre estes objetos, chamados morfismos.

<adicionar notações matemáticas e diagramas>

B. *Functor – Categories e Morfismos*

Um functor F consiste em um morfismo entre duas categories [7]. Estes morfismos, chamados homomorfismos, tem como característica a preservação de estrutura de F. Assim como categories, functors devem satisfazer os axiomas de associatividade e identidade.

<adicionar notações matemáticas e diagramas>

C. *Monad – Endofunctor e Transformações Naturais*

Endofunctors equipados com as natural transformations unit e multiplication [7] são chamados de monad. Define-se endofunctors como functors cujos morfismos dão-se em uma mesma category. Além disso, natural transformations são morfismos entre functors, tendo como principal característica a preservação de estrutura. Como todas as estruturas já mencionadas, monads também devem satisfazer as propriedades de associatividade e identidade.

<adicionar notações matemáticas e diagramas comutativos>

D. *Arrow – Kleisli Triples*

A partir de um monad podemos construir kleisli triples. Podemos pensar em kleisli triples como uma outra forma de representar sintaticamente um monad. Similar a monads, estas também obedecem as propriedades de associatividade e comutatividade.

<adicionar notações matemáticas e diagramas comutativos>

REFERÊNCIAS

[1] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys (CSUR)*, vol. 21, no. 3, pp. 359–411, 1989.

[2] K. Louden *et al.*, *Programming Languages: Principles and Practices*. Cengage Learning, 2011.

[3] G. Michaelson, *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation, 2011.

[4] B. Goetz, R. Forax, D. Lea and B. Lee, "State of the lambda," Sept 2013, White Paper. [Online]. Available: http://cr.openjdk.java.net/ briangoetz/lambda/lambda-state-final.html

[5] C. Hunt and B. John, *Java Performance*. Prentice Hall Press, 2011.

[6] B. Goetz, "Translation of lambda expressions," Apri 2012, White Paper. [Online]. Available: http://cr.openjdk.java.net/ briangoetz/lambda/lambda-translation.html

[7] S. Mac Lane, *Categories for the Working Mathematician*, ser. Graduate Texts in Mathematics. Springer-Verlag, 1971, no. 5.

[8] D. I. Spivak, *Category Theory for the Sciences*. The MIT Press, 2014, no. 1.

[9] S. P. Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell," in *Engineering theories of software construction*. Press, 2001, pp. 47–96.

[10] E. Moggi, "Notions of computation and monads," *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, Jul. 1991.