

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 ideia 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 (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 colaterais (side-effects). Também destaca-se o princípio de imutabilidade, onde o valor de uma variável é determinado em sua criação, não permitindo novas atribuições posteriormente. Ao expressar um programa em uma linguagem funcional obtém-se uma maneira concisa de solucionar problemas, sendo que este constitui-se de operações e objetos atômicos e regras gerais para sua composição [3].

Tais propriedades são apreciadas nos tempos atuais, visto que a Lei de Moore nos fornece cada vez mais núcleos, não necessariamente núcleos mais rápidos [4]. Em programas não-determinísticos, múltiplas threads podem alterar os dados representados por objetos imutáveis sem ocasionar os diversos problemas já conhecidos como dead locks e race conditions. Além do thread safety oferecido pela propriedade de imutabilidade, o conceito de funções transparentes referencialmente, ou seja, funções que não utilizam variáveis de estado, também oferecem vantagens para a concorrência com múltiplas threads podendo invocar tais funções esperando sempre o mesmo retorno, dados os mesmos argumentos.

III. JAVA LAMBDA EXPRESSIONS

Na última década muitos problemas encontrados – como enviar não só dados, mas também comandos através de redes – já foram solucionados em linguagens que suportam o paradigma funcional [5]. Assim, linguagens multi-paradigma têm adicionado suporte à estas mesmas estruturas, aumentando sua flexibilidade e ganho para com os desenvolvedores. O suporte a lambda expressions em Java não tem como objetivo apenas substituir Anonymous Inner Classes, mas também ser capaz de trazer os benefícios deste paradigma ao ponto de incrementar o ecossistema da linguagem.

A. Lambda Expressions e Anonymous Inner Classes

Ao fornecer suporte a funções de primeira classe, também chamadas de lambda expressions ou closures ¹, a linguagem Java habilita a substituição de anonymous inner classes (AIC) de forma transparente. Conforme a listagem 1, em Java a ordenação de inteiros pode ser implementada a partir de uma AIC em conjunto do método sort da classe Arrays.

```
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);  
    }  
});
```

Listagem 1: Sort - Anonymous Inner Class

Neste trecho de código o método *sort* recebe como primeiro argumento um array de inteiros (já declarado na variável *integers*) e como segundo qualquer instância que satisfaça o contrato de *Comparator*. Assim, para satisfazer o segundo argumento, instância-se uma AIC através da palavra reservada **new** que implementa o método *compare* declarado no contrato.

¹Lambda expressions ou closures são funções que não exigem vínculos de classe, como exemplo podendo ser atribuída a uma variável. Com esta característica, uma função atua como dado, ou seja, pode ser passada como argumento para outras funções.

Entretanto, com closures o mesmo método *sort* pode ter seus argumentos simplificados. Conforme demonstrado na listagem 2, ao invés de instanciar uma AIC, uma lambda expression é passada como segundo argumento, removendo a necessidade de instanciação de uma classe e a implementação de um contrato imposto por *Comparator*.

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

Listagem 2: Sort - Lambda Expression

De fato, apesar de expressarem o mesmo comportamento a nível de código, ambas funcionalidades possuem diferentes implementações sob a Máquina Virtual Java (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 classloader e seus construtores invocados pela máquina virtual. Ambas etapas consomem memória, tanto heap [6] para alocação de objetos quanto permgen².

Diferentemente de AIC, lambdas postergam a estratégia de compilação para em tempo de execução, utilizando a instrução *invokedynamic* [7]. 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 lambda expression é acessada, ao contrário de AIC que acessa sua própria instância.

Além destes benefícios, lambda expressions possibilitam a implementação de novas bibliotecas que trazem consigo as propriedades do paradigma funcional. São suficientemente poderosas para expressar as mesmas construções, sem os encargos impostos pelas antigas implementações. Além disso, a adoção do paradigma funcional torna os usuários aptos a aderirem a novas estruturas advindas da Category Theory, como a classe *Optional*, disponível na última versão da linguagem.

IV. CATEGORY THEORY E SUAS ESTRUTURAS

A Category Theory (CT) foi inventada no início dos anos 1940 por Samuel Eilenberg e Saunders Mac Lane [8] como uma ponte entre os diferentes campos da topologia e álgebra [9]. Afim de demonstrar as relações entre estruturas e sistemas matemáticos [10], a CT estabelece uma linguagem formal capaz de encontrar aplicabilidade em várias áreas da ciência. Tal como Group Theory³ abstrai a ideia do sistema de permutações como simetrias de um objeto geométrico, a CT manifesta-se como um sistema de funções entre conjuntos de objetos [11].

Conforme a figura 1, os conjuntos de objetos são representado por A , B e C . Nesta mesma estrutura, as funções

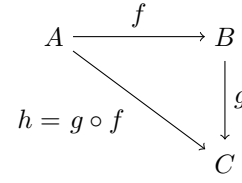


Figura 1: Funções entre coleções de objetos

f e g denotam os morfismos entre os diferentes conjuntos de objetos, tal que $f : A \rightarrow B$ e $g : B \rightarrow C$. A função h , definida por $h : A \rightarrow C$ dá-se como produto da composição de f e g . A composição de funções, conforme a seção II e expressa através de h na figura 1, firma uma ligação entre o paradigma funcional e a organização estrutural derivada da CT.

A partir desta ligação, Moggi em seus trabalhos [12, 13] introduziu o conceito de monad em Haskell. Este foi capaz de fornecer os mecanismos necessários para capacitar a solução de um dos problemas fundamentais da linguagem, pertencente ao conjunto The Awkward Squad [14]. A partir de outros conceitos como functors, categories, arrows, o trabalho de Moggi foi incrementado criando novas estruturas que estão presentes nas versões mais recentes da linguagem.

A. Category – Objetos e Morfismos

Uma category consiste em uma coleção de coisas, todas relacionadas de algum modo. As coisas são nomeadas de objetos e as relações de morfismos [9].

Definição: conforme [9, 10], uma category C é definida como:

- (a) uma coleção $Ob(C)$, contendo os objetos de C ;
- (b) para cada par $a, b \in Ob(C)$, um conjunto $Hom_C(a, b)$ chamado de morfismos de a para b ;
- (c) para cada objeto $a \in Ob(C)$, um morfismo de *identidade* em a denotado por $id_a \in Hom_C(a, a)$;
- (d) para cada três objetos $a, b, c \in Ob(C)$, uma função de composição $\circ : Hom_C(b, c) \times Hom_C(a, b) \rightarrow Hom_C(a, c)$;

Dado os objetos $a, b \in Ob(C)$, denota-se o morfismo $f \in Hom_C(a, b)$ por $f : a \rightarrow b$; onde a é o domínio e b o contradomínio.

Estas operações em C devem satisfazer os seguintes axiomas:

(*Identidade*) Para todo objeto $a, b \in Ob(C)$ e todo morfismo $f : a \rightarrow b$, tem-se $id_b \circ f = f = f \circ id_a$;

(*Associatividade*) Sejam os objetos $a, b, c, d \in Ob(C)$ e os morfismos $f : a \rightarrow b$, $g : b \rightarrow c$ e $h : c \rightarrow d$. Então $(h \circ g) \circ f = h \circ (g \circ f)$.

²Área de memória limitada separada da heap chamada Permanent Generation que possui a função de armazenar objetos de geração permanente como metadados, classes e métodos.

³A Group Theory estuda as estruturas algébricas chamadas Group. Um Group é um conjunto de elementos finito ou infinito associado a uma operação binária, como por exemplo a adição ou multiplicação.

Ambos axiomas podem ser representados pelos diagramas comutativos das figuras 2 e 3.

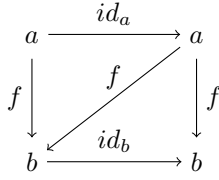


Figura 2: Identidade

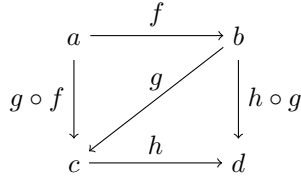


Figura 3: Associatividade

B. Functor – Morfismos entre Categories

Um functor é um mapeamento entre duas categorias de tal modo que o domínio, contradomínio, objetos e morfismos são preservados [11].

Definição: para as categorias C e D , um functor $F : C \rightarrow D$ com domínio C e contradomínio D consiste em duas funções relacionadas [10]. São elas:

- (a) a função de objeto F que atribui cada objeto $a \in Ob(C)$ para um objeto $F(a) \in Ob(D)$;
- (b) e a função de flecha (também chamada F) que atribui cada morfismo $f : a \rightarrow b \in Hom_C(a, b)$ para um morfismo $F(f) : F(a) \rightarrow F(b) \in Hom_D(a, b)$.

Tal que os seguintes axiomas são satisfeitos:

(Identidade) Para todo objeto $a \in Ob(C)$ existe um morfismo $F(id_a) = id_{F(a)}$ que preserva a identidade;

(Associatividade) Sejam os objetos $a, b, c \in Ob(C)$ e os morfismos $f : a \rightarrow b$ e $g : b \rightarrow c$. Então $F(g \circ f) = F(g) \circ F(f)$.

Estas funções capazes de preservar as características das categories C e D também podem ser ilustradas a partir da figura 4.

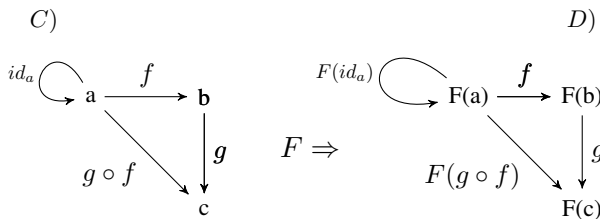


Figura 4: Functor F

C. Monad – Endofunctor e Transformações Naturais

Podendo ser visto como um padrão estrutural que ocorre diversos ramos da matemática [9], a construção fundamental chamada monad estrutura-se a partir de um endofunctor ⁴ e

⁴Um functor que mapeia uma categoria para ela mesma.

das transformações naturais ⁵ de identidade e multiplicação.

Definição: o monad $T = (T, \eta, \mu)$ em uma category C consiste em:

- (a) um endofunctor T , tal que $T : C \rightarrow C$;
- (b) nas transformações naturais:

(Identidade) $\eta : id_C \rightarrow T$;

(Multiplicação) $\mu : T^2 \rightarrow T$, onde $T^2 = T \circ T$.

Fazendo com que os diagramas 5 e 6 comutem, respeitando os axiomas de identidade e associatividade.

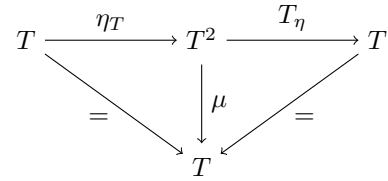


Figura 5: Identidade à esquerda e direita

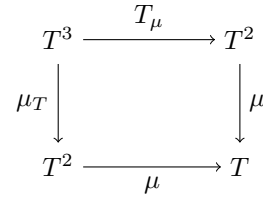


Figura 6: Associatividade

D. Arrow – Kleisli Triples

Triplas de Kleisli são equivalentes a mónades, oferecendo uma outra forma representação sintática. Nesta, objetos ainda estão contidos na categoria original, mas seus morfismos fazem parte da Categoria de Kleisli, habilitando a de composição destes.

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. Loudon *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 *et al.*, “State of the lambda,” Sept 2013, White Paper. [Online]. Available: <http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>

⁵Mapeamento entre dois functors que possuem o mesmo domínio e contradomínio, tal que satisfaça a condição de naturalidade.

- [5] R. Fischer, *Java Closures and Lambda*, 1st ed. Apress, 2015.
- [6] C. Hunt and B. John, *Java Performance*. Prentice Hall Press, 2011.
- [7] B. Goetz, "Translation of lambda expressions," Apr 2012, White Paper. [Online]. Available: <http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html>
- [8] S. Eilenberg and S. MacLane, "General theory of natural equivalences," *Transactions of the American Mathematical Society*, pp. 231–294, 1945.
- [9] D. I. Spivak, *Category Theory for the Sciences*. The MIT Press, 2014, no. 1.
- [10] S. Mac Lane, *Categories for the Working Mathematician*, ser. Graduate Texts in Mathematics. Springer-Verlag, 1971, no. 5.
- [11] S. Awodey, *Category Theory - Oxford Logic Guides*, 2nd ed. Oxford University Press, 2010, vol. 52.
- [12] E. Moggi, "Computational lambda-calculus and monads," pp. 14–23, 1989.
- [13] E. Moggi, "Notions of computation and monads," *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, Jul. 1991.
- [14] 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.