

Teoria das Categorias Mónades e Flechas em Java

Pablo M. Parada¹, Mozart L. Siqueira²

Ciência da Computação

Centro Universitário La Salle - Unilasalle

E-mail: pablo.paradabol@gmail.com¹, mozarts@unilasalle.edu.br²

Resumo—<escrever>

Index Terms—<escrever>

I. INTRODUÇÃO

<escrever>

II. 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 será 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, há também o conceito de funções transparentes referencialmente, ou seja, funções que não utilizam variáveis de estado. Em ambientes distribuídos onde a execução é subdivida em diferentes threads, a transparência referencial garante sempre o mesmo retorno, dados os mesmos argumentos.

O paradigma funcional expressa programas através de composições mantendo a imutabilidade e a transparência referencial, portanto apresenta características importantes para os tempos atuais. A complexidade intrínseca à ambientes distribuídos

é reduzida. Programas completos são vistos como apenas uma aplicação de função.

III. JAVA LAMBDA EXPRESSIONS

Na última década muitos dos 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 closures¹, 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

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.

new que implementa o método *compare* declarado no contrato.

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.

Dessa forma, o suporte a lambda expressions traz benefícios para os usuários da linguagem. Tal funcionalidade está além de uma mera substituição, pois acrescenta um novo paradigma no ecossistema Java.

IV. TEORIA DAS CATEGORIAS E SUAS ESTRUTURAS

A Teoria das Categorias (TC) 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 TC estabelece uma linguagem formal capaz de encontrar aplicabilidade em várias áreas da ciência. Por volta dos anos 1980, computação e TC passaram a ser consideradas áreas correlatas de estudo [11].

Aplicações do modelo categorial ocorrem na composição de funções encorajada pelo paradigma funcional. Além do mais, em linguagens de programação, o estudo dos tipos pode ser representado através de categorias. Muitos modelos computacionais que fazem uso de estruturas de dados como grafos podem ser generalizados para categorias de grafos. Portanto,

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.

tais aplicações demonstram a capacidade de abstração e a importância da TC para a computação.

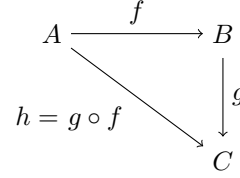


Figura 1: Funções entre coleções de objetos

Tal como a Teoria dos Grupos³ abstrai a ideia do sistema de permutações como simetrias de um objeto geométrico, a TC manifesta-se como um sistema de funções entre conjuntos de objetos [12]. Esta abstração pode ser vista a partir da Figura 1, onde os conjuntos de objetos são representado por A , B e C . Nesta mesma estrutura, f e g denotam os morfismos entre os diferentes conjuntos de objetos, tal que $f : A \rightarrow B$ e $g : B \rightarrow C$. Por fim, h expressa a ideia de composição, sendo produto da união dos morfismos f e g .

A TC nasceu como uma ferramenta matemática, com propósito de estudar a relação objetos e morfismos. Contudo, tal conceito tornou-se uma ferramenta utilizada por diversas áreas da ciência. Na computação, diversos problemas são abstraídos através de estruturas da TC, pois ajudam na sua resolução.

A. Categoria – Objetos e Morfismos

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

Definição [9, 10]: define-se a categoria C 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 $id_a \circ f = f = f \circ id_b$;

³Teoria que estuda as estruturas algébricas de grupos. Um grupo é formado por um conjunto de elementos finito ou infinito associado a uma operação binária, como por exemplo a adição ou multiplicação.

(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 $(h \circ g) \circ f = h \circ (g \circ f)$.

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

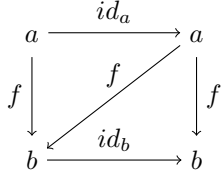


Figura 2: Identidade



Figura 3: Associatividade

B. Funtor – Morfismos entre Categorias

Um funtor é um mapeamento entre duas categorias de tal modo que o domínio, contradomínio, objetos e morfismos são preservados [12].

Definição [9, 10]: para as categorias C e D , um funtor $F : C \rightarrow D$ com domínio C e contradomínio D consiste em duas funções relacionadas. 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 categorias C e D também podem ser ilustradas a partir da Figura 4.

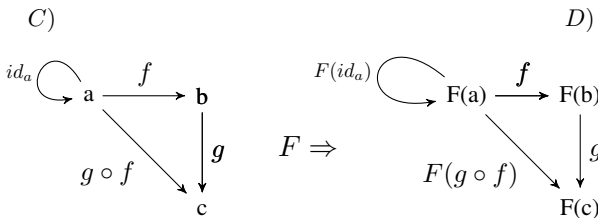


Figura 4: Funtor F

C. Mônade – Endofuntor e Transformações Naturais

Podendo ser visto como um padrão estrutural que ocorre diversos ramos da matemática [9], a construção fundamental mônade (também chamada de tripla) estrutura-se a partir de um endofuntor⁴ e das transformações naturais⁵ de identidade e multiplicação.

Definição [10, 12]: dada a mônade $T = (T, \eta, \mu)$ em uma categoria C consiste em:

- (a) um endofuntor 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 da Figura 5 e Figura 6 comutem, respeitando os axiomas de identidade (esquerda e direita) e associatividade.



Figura 5: Identidade à Esquerda e à Direita



Figura 6: Associatividade

D. Flechas – Categoria de Kleisli

A partir de uma tripla obtemos a Categoria de Kleisli (CK), capaz de representar a mesma estrutura monádica através de uma sintaxe diferente [13]. Mantendo definições similares as já apresentadas em IV-A, a CK destaca-se por abstrair a estrutura de objetos e morfismos sendo capaz compor o endofuntor subjacente.

Definição [10, 13, 14]: dada a tripla (T, η, μ) sob a categoria C , então define-se C_T como:

- (a) cada objeto $a \in Ob(C)$, um novo objeto a_T ;

⁴Um funtor que mapeia uma categoria para ela mesma.

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

- (b) cada morfismo $f : a \rightarrow T_b$, um novo morfismo $f^* : a_T \rightarrow b_T$.

Dado os morfismos $f^* : a_T \rightarrow b_T$, $g^* : b_T \rightarrow c_T$ e $h : c \rightarrow d_T$ e o operador de extensão $-^*$, então:

$$(Composição) \quad g^* \circ f^* = (\mu_c \circ T(g) \circ f)^*.$$

Similar as outras estruturas desta seção, C_T deve obedecer as seguintes leis:

$$(Identidade \ à \ Esquerda) \quad f^* \circ \eta_a = f;$$

$$(Identidade \ à \ Direita) \quad \eta_a^* \circ h = id_{T(a)} \circ h;$$

$$(Associatividade) \quad (g^* \circ (f^* \circ h)) = (g^* \circ f)^* \circ h.$$

V. APLICAÇÃO EM HASKELL

As funções do paradigma funcional podem ser vistas como morfismos na categoria dos tipos. Percebendo estas relações, Wadler [15–18] utilizou o conceito de mónade para estruturar programas puramente funcionais em Haskell. Assim, o primeiro problema pertencente ao conjunto Awkward Squad foi solucionado [19].

A introdução da mónade para E/S (entrada e saída) padronizou a maneira de executar estas ações e encapsular seus efeitos colaterais. Assim, outras mónades foram adicionadas à linguagem, estendendo as bibliotecas padrões para fornecer suporte a exceções, nulidade, concorrência, etc. Logo, mónades trouxeram utilidade para a linguagem, pois confrontaram o Awkward Squad [19].

Com a rápida adoção de mónades para a solução de problemas envolvendo E/S, constatou-se que outros conceitos da TC também poderiam ser aproveitados. Qualquer estrutura recursiva (listas, mapas, árvores, grafos) que possa ser iterada é representada por um funtor. A categoria de Haskell, chamada Hask, preocupa-se em tratar tipos como objetos e funções como morfismos, fornecendo funções de composição e identidade.

Type classes definem comportamentos genéricos que podem ser implementados por um conjunto variado de tipos. Esta funcionalidade tornou a implementação das estruturas previamente mencionadas disponíveis para qualquer tipo de dado, dependendo apenas da implementação de suas instâncias [20].

Conforme a Typeclassopedia [21], a Figura 7 demonstra as relações entre as type classes, identificadas por flechas tracejadas e pontilhadas:

- (a) (*Tracejadas*) Determinam relações de é-um, ou seja, se existe uma flecha de A para B , então todo B é um A ;
- (b) (*Pontilhadas*) Indicam algum tipo de relação, como por exemplo a equivalência entre *Monoid* e *MonadPlus*.

A implementação de uma type class pode ser visualizada na Listagem 3, onde a primeira declaração incorpora uma

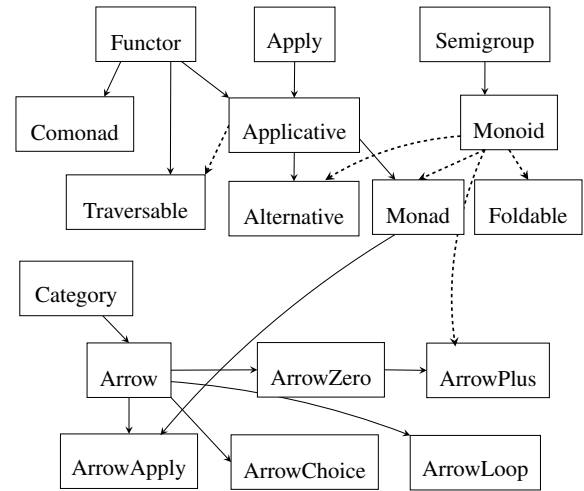


Figura 7: Hierarquia de type classes⁶

nova classe com nome *Eq*, uma variável de tipo a e um comportamento explicitado pela função *equals*.

```
class Eq a where
  equals :: a -> a -> Bool
```

Listagem 3: Type class *Eq*⁷

Na Listagem 4, instancia-se *Eq* para os tipos *Int* e *Char*. A primeira instancia declara que *Int* pertence a *Eq*, e que a implementação de igualdade de inteiros é dada por *primEqInt*. Similarmente, *Char* pertence *Eq* e fornece a implementação de igualdade por *primEqChar*. Logo, como a é um parâmetro de tipo, as assinaturas de *primEqInt* e *primEqChar* são tipadas $Int \rightarrow Int \rightarrow Bool$ e $Char \rightarrow Char \rightarrow Bool$ consecutivamente [22].

```
instance Eq Int where
  equals = primEqInt

instance Eq Char where
  equals = primEqChar
```

Listagem 4: Instancias *Eq*⁷

Com a instancia de *Eq Int* é possível comparar dois inteiros tal que *equals*(1 + 1, 2) retorna *True*; ou *equals*(1 + 1, 0) retorna *False*. De maneira similar, com *Eq Char* é possível comparar dois caracteres, onde *equals*('a', 'a') retorna *True*; ou *equals*('a', 'b') retorna *False*.

⁶Fonte: <https://wiki.haskell.org/Typeclassopedia>. Acessado em 17/09/2015.

⁷Fonte: C. V. Hall *et al.*, Type Classes in Haskell, p. 3, 1996.

A. Monad

Para a programação funcional, mónades oferecem um contexto computacional que encapsula efeitos colaterais. Além disso, tal contexto monádico permite o encadeamento de funções que operam sob um determinado tipo. Em Haskell, esta abstração é expressa através da type class *Monad*.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: (>>) m a -> m b -> m b
  return :: a -> m a
```

Listagem 5: Type class *Monad*⁸

Conforme a Listagem 5, a type class *Monad* consiste em um tipo construtor *m* e as operações *>>=*, *>>* e *return*. A operação *>>* existe apenas como um facilitador para a composição de funções no contexto *m*. *>>=* e *return* são equivalentes as transformações naturais de multiplicação e identidade.

A operação *return* encapsula o tipo *a* no contexto monádico *m a* e retorna este como resultado. A próxima é *bind*, declarada simbolicamente por *>>=*, que habilita o encadeamento de funções. *Bind* aplica a função *a* → *m b* no contexto monádico *m a*, retornando uma nova instancia de *Monad* do tipo *m b*.

Todas instâncias de *Monad* devem seguir as leis monádicas de associatividade e identidade. Estes axiomas são demonstrados na Listagem 6.

```
return a >>= k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Listagem 6: Leis para *Monad*⁸

A possibilidade de encadear funções com *>>=* é tão importante que diversas linguagens funcionais oferecem suporte sintático para mónades. Em Haskell, esta sintaxe especial é chamada de **do notation** [23].

```
putStr "x: " >>
getLine >>= \l ->
return (words l)
```

Listagem 7: Encadeando Funções com Bind e Then⁸

Na Listagem 7 as funções *putStr* e *getLine* são encadeadas pelos operadores *>>* e *>>=*. Estas mesmas funções podem ser encadeadas utilizando a utilizando o suporte sintático de **do notation**, conforme a Listagem 8.

```
do
  putStr "x: "
  l <- getLine
  return (words l)
```

Listagem 8: Encadeando Funções com Do Notation⁸

Com esta notação, um código pertencente ao paradigma funcional é escrito de forma imperativa, ou seja, a computação é declarada passo a passo. Assim, obtém-se uma sintaxe simplificada para expressar efeitos colaterais em linguagens puramente funcionais como Haskell.

B. Arrow

Conforme o exemplo de programação tácita ¹¹ da Listagem 9, Hughes [24] constrói uma função que conta o numero de ocorrências de uma palavra em um *String*.

```
count w = length . filter (==w) . words
```

Listagem 9: Função *count*¹²

A função *count* é implementada pela união de *length*, *filter*, *==* e *words*. Por último mas não menos importante, a função de composição *(.)* atua na implementação de *count* como um operador de ligação entre as funções menores. Entretanto, funções com efeitos colaterais não são compostas tão facilmente. Conforme a Listagem 10, *readFile* retorna um *String* encapsulado pela mónade *IO*. Em contrapartida, *count* espera um *String* como argumento, não um tipo construtor *m*.

```
readFile :: FilePath -> IO String
```

Listagem 10: Funções de E/S¹²¹⁰

Não há como por exemplo utilizar *(.)* para unir *readFile* e *count*, ou seja, não é possível ler um arquivo do disco e contar a ocorrência de uma determinada palavra. Entretanto, a união destas é desejável. Sendo assim, para permitir a composição destes tipos de funções, Hughes introduziu a type class *Arrow*. Na Listagem 11, *Arrow* denota uma computação pelos tipos *b*, *c* e *d*, um tipo construtor *a* e as operações *arr* e *>>>*.

⁸Fonte: S. Marlow *et al.*, Haskell 2010 Language Report, p. 25–81, 2010.

⁹Fonte: A. Courtney e E. Conal, Genuinely functional user interfaces, p. 5, 2001.

¹⁰Fonte: <http://hackage.haskell.org/package/base-4.8.1.0/docs/Prelude.html>. Acessado em: 26/09/2015.

¹¹Um paradigma de programação no qual as funções não identificam seus argumentos, em vez disso são construídas a partir de funções combinatórias que manipulam os argumentos.

```
class Arrow a where
  arr  :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
```

Listagem 11: Type class *Arrow*⁹

Destas, a mais comum é *arr*, declarada com a assinatura $(b \rightarrow c) \rightarrow a b c$. Dada qualquer função com entrada *b* e saída *c*, *arr f* constrói uma instância de *Arrow* a partir de *f* [25].

A segunda operação é definida simbolicamente por $(>>>)$ com assinatura $a b c \rightarrow a c d \rightarrow a b d$. Recebe como argumento uma instância definida por *a c d* e retorna *a b d*. Logo, $(>>>)$ aplica sequencialmente as funções passadas para *arr*, gerando uma computação de entrada *b* e saída *d* encapsuladas em uma instância de *Arrow*.

Tem-se na Listagem 12 uma instancia de *Arrow* que fornece a composição de funções ordinárias. Nesta, *arr* é implementada pela função de identidade *id*, disponível no pacote *Prelude*. Na operação de composição $(>>>)$, utiliza-se $(.)$ e *flip* [24]. Assim, define-se o tipo *Arrow* (\rightarrow) , instancia de *Arrow* para funções ordinárias. Sua assinatura é dada por $a \rightarrow b$, sendo *a* e *b* tipos quaisquer.

```
instance Arrow (->) where
  arr  = id
  (>>>) = flip (.)
```

Listagem 12: Arrow - Composição de Funções¹²

Além disso, a definição do tipo *Kleisli* para *Arrow* na Listagem 13 e 14 provê a composição de mónades. Os tipos de entrada e saída são definidos explicitamente em *Kleisli*, diferentemente de *Monad*. Portanto, *Arrow (Kleisli m)* generaliza *Monad* [24].

```
newtype Kleisli m a b = Kleisli {
  runKleisli :: a -> m b
}
```

Listagem 13: Definição do tipo *Kleisli*¹²

```
instance Monad m => Arrow (Kleisli m) where
  ...

  first (Kleisli f) = Kleisli (\(a,c) ->
    do b <- f a
    return (b,c))
```

Listagem 14: Instância para *Monad*¹²

Apesar de possuir nove axiomas, na Listagem 15 estão listados apenas os necessários para as operações de composição e identidade [26].

```
arr id >>> f = f
f >>> arr id = f
(f >>> g) >>> h = f >>> (g >>> h)
arr (g . f) = arr f >>> arr g
...
```

Listagem 15: Leis para *Arrow*¹³

VI. APLICAÇÃO EM JAVA

A linguagem Java originou-se a partir de um projeto de pesquisa, tendo como objetivo de desenvolver uma plataforma capaz de operar em tempo real. Em sua criação, decisões arquiteturais e de design foram embasadas em linguagens como SmallTalk, Eiffel, Objective C e Cedar/Mesa [27]. Para ser considerada uma linguagem orientada a objetos, Java suporta encapsulamento, polimorfismo e herança desde sua elaboração.

Desde sua quinta versão, tendo como base o trabalho de Bracha *et al.* [28, 29], Java conta com tipos genéricos. A abstração de tipos nos permite utilizar tipos parametrizáveis, generalizando implementações de interfaces, classes e métodos. Com isto o compilador é capaz de checar os tipos em tempo de compilação, evitando erros de conversão e em tempo de execução.

Recentemente obteve-se o suporte a funções anônimas. Estas foram adicionadas a partir do Lambda Project [4], na versão 8 da JDK. De extrema importância, Lambda Expressions acrescentam novas características compatíveis com as já existentes em linguagens funcionais. Logo, contando com as funcionalidades já citadas e as recém adicionadas na plataforma, Java é capaz de fornecer suporte a mónades e flechas.

A. Problema dos Tipos Construtores

Construtores de classe constroem objetos, respectivamente, tipos construtores constroem tipos. Tipos construtores possuem aridade, ou seja, recebem tipos como argumento [30]. Se um tipo construtor não receber argumentos, este é chamado de tipo construtor nulário. Um tipo qualquer *A* origina-se a partir de um construtor nulário, assim como um tipo $C(A)$ é aplicado para construtores de aridade *n*.

Tipos genéricos são equivalentes a construtores unários, incapazes de representar tipos construtores com aridade *n*. Tendo como exemplo o mesmo tipo $C(A)$ anterior, este poderia ser expresso por genéricos se *C* fosse estaticamente definido. Portanto, para qualquer *C* definido estaticamente

¹²Fonte: J. Hughes, Programming with Arrows, p. 73–81, 2005.

¹³Fonte: S. Lindley e P. Wadler, Idioms are oblivious, arrows are meticulous, monads are promiscuous, p. 97–98, 2011.

como *List*, *Set*, *Queue*, *Map*, e assim em diante, é possível parametrizar *A* com genéricos.

Entretanto, para a construção da interface monádica é necessário expressar um tipo construtor de aridade *n*. A declaração de um tipo *Monad(M(A))* utilizando tipos genéricos é inválida, ocasiona um erro em tempo de compilação. Logo, a declaração de interface abaixo não pode ser expressa na linguagem.

```
public interface Monad<M<A>> {}
```

Listagem 16: Interface *Monad* inválida

Afim de solucionar esta inconsistência, na Listagem 17 propõe-se o tipo *Kind(A, K)*. Sendo capaz de simular tipos construtores, este possui os parâmetros de tipo genérico *A* e *K*.

- (a) (*A*) este primeiro parâmetro genérico é equivalente ao valor encapsulado *A* na declaração *Monad(M(A))*.
- (b) (*K*) determina uma restrição de herança, na qual *K* herda *Kind* e parametriza este com um tipo curinga e o próprio *K* que representa *Kind*.

```
public interface Kind<A, K extends Kind<?, K>> {}
```

Listagem 17: Interface *Kind*

A hierarquia imposta na herança de *K* e *Kind* limita a simulação de tipos construtores reais. Em outras linguagens, estes apenas são preenchidos por tipos, sem necessariamente impor uma classificação. Contudo, subclasses que implementem o contrato de *Kind* mantém o controle do tipo mais interno *A* e do mais externo *K*. Além disso, estas também podem ser utilizadas como tipos construtores pois herdam as características de *Kind* após implementá-lo.

B. Implementação de *Monad*

A interface *Monad*, conforme 18, conta com as operações monádicas de identidade e multiplicação por meio de seu construtor de classe e do método *flatMap*. Afim de tornar *M* um tipo construtor, *Monad* implementa *Kind* e parametriza este nos tipos *A* e *M*. Também é importante salientar a implementação de *Functor* que mantém consistente a hierarquia dessas construções e garante a implementação de *map*.

Em sua declaração, *M* herda *Kind*, porém não especializa o valor deste primeiro tipo. Durante a declaração de interface ou classe, apenas os tipos genéricos deste escopo estão disponíveis, e desta forma não é possível determinar qual o valor do primeiro parâmetro de tipo de *Kind*. Como existe apenas *A* e os métodos da classe podem retornar outros

valores, determina-se que *Kind* é parametrizado pelo operador coringa.

Para a declaração de *flatMap* utilizam-se genéricos no escopo de método. Os tipos *B* e *M(B)* representam a saída da operação natural de multiplicação de mónade. Com estes declarados, a função *f* que será aplicada neste contexto consegue expressar a entrada *A* e saída *M(B)*. Como na declaração de classe *Kind* foi parametrizado com o operador coringa, agora é possível especializar o retorno de *flatMap* por *Monad(M(B))*.

O método *map* é herdado de *Functor* e seu tipo de retorno é especializado em *Monad*. Neste, utiliza-se o mesmo modo de declaração de *flatMap* com genéricos no escopo de método. Sua declaração é mais simples, pois a função *f* retorna um tipo simples *B*. Por fim, *M* é utilizado no retorno de *map* onde declara-se *Monad(M(B))*.

Diversas estruturas que utilizam o conceito do contexto monádico podem implementar o contrato *Monad* da Listagem 18. Os métodos *map* e *flatMap* e os parâmetros de tipo exigidos pela interface visam garantir consistência entre suas implementações. Para demonstrar os benefícios de *Monad*, na Listagem 19, 20 e 21 a interface *Maybe* e suas subclasses *Just* e *Nothing* são implementadas.

```
public class Just<A> implements Maybe<A> {  
    ...  
    public Just(A value) {  
        this.value = value;  
    }  
    public static <A> Just<A> pure(A a) {  
        return new Just<>(a);  
    }  
    ...  
}
```

Listagem 20: Classe *Just*

O tipo *Maybe* é uma mónade que encapsula um valor opcional. Esta pode conter valor e ser uma instância de *Just(A)*, ou estar vazia e corresponder a *Nothing*. Os métodos *map* e *flatMap* dispõem o encadeamento de funções no contexto de *Maybe*. Os construtores e métodos utilitários como *pure* determinam como dá-se a instanciação.

```
public class Nothing<A> implements Maybe<A> {  
    public Nothing() {}  
    ...  
}
```

Listagem 21: Classe *Nothing*

```
public interface Monad<A, M extends Kind<?, M>> extends Functor<A>, Kind<A, M> {
    <B, MB extends Kind<B, M>> Monad<B, M> flatMap(Function<? super A, MB> f);
    <B> Monad<B, M> map(Function<? super A, ? extends B> f);
}
```

Listagem 18: Interface *Monad*

```
public interface Maybe<A> extends Monad<A, Maybe<?>> {
    ...
    default <B, MB extends Kind<B, Maybe<?>>> Maybe<B> flatMap(Function<? super A, MB> f) {
        if (isDefined()) return (Maybe<B>) f.apply(get());
        else return new Nothing<>();
    }
    default <B> Maybe<B> map(Function<? super A, ? extends B> f) {
        if (isDefined()) return Just.pure(f.apply(get()));
        else return new Nothing<>();
    }
    ...
}
```

Listagem 19: Interface *Maybe*

Na Listagem 22 demonstra-se a utilização de *Maybe* e suas subclasses para evitar exceções em tempo de execução. O método *divide* recebe dois argumentos do tipo *double* e garante que divisões por zero não ocorram, retornando *Just(Double)* ou *Nothing*. No retorno deste método, encadeia-se uma lambda expression trivial de soma com *flatMap*. Portanto, se *n* assumir valor zero, o retorno de *flatMap* é *Nothing*, caso contrário teremos o valor inicial de *a* dividido por *b* e acrescentado em dois.

```
public Maybe<Double> divide(double a, double b) {
    if (b == 0) return new Nothing();
    else return Just.pure(a / b);
}
...
divide(10, n).flatMap(x -> Just.pure(x + 2))
```

Listagem 22: Exemplo de utilização de *Maybe*

Consequentemente, *Maybe* possibilita que os axiomas de identidade e associatividade de *Monad* sejam validados. Na Listagem 23, os métodos *leftIdentity*, *rightIdentity* e

associativity representam as leis que regem a estrutura e garantem que estas são válidas para *Maybe*, garantindo a fidelidade da interface *Monad*.

```
public void leftIdentity() {
    assertEquals(Just.pure(a).flatMap(f),
        f.apply(a));
}

public void rightIdentity() {
    assertEquals(m.flatMap(Just::pure), m);
}

public void associativity() {
    assertEquals(m.flatMap(f).flatMap(g),
        m.flatMap(x -> f.apply(x).flatMap(g)));
}
```

Listagem 23: Leis de *Monad*

a) *Identidade* - *leftIdentity* e *rightIdentity*: encapsular um valor em um contexto monádico não deve apresentar alterações no valor ou na estrutura da mónade.

b) *Associatividade - associativity*: a ordem na qual as funções são encadeadas em um contexto monádico não deve importar.

Todas as leis estão implementadas considerando a igualdade de objetos descrita por *equals* e *hashCode* [27]. O método *assertEquals* recebe dois argumentos como valores comparativos e a equivalência dos objetos deve resultar em um tipo booleano verdadeiro, caso contrário a as leis são invalidadas.

C. Problema da Generalização de Arrow

Na declaração de *Arrow* na Listagem 24, a interface apresenta três parâmetros de tipo. Como primeiro tipo genérico, *A* representa um tipo construtor por ser subtipo de *Kind*. O restante, *B* e *C*, apenas explicitam os tipos de entrada e saída de *Arrow*.

Assim como *Monad*, subclasses de *Arrow* devem utilizar seus construtores para incluir $B \rightarrow C$ em seus contextos. Além disso, o método *andThen* define a composição de *Arrow* por $A(B, D) = A(B, C) \circ A(C, D)$.

```
interface Arrow<A extends Kind<?, ?>, B, C> {
    <D, AD extends Kind<D, ?>> Arrow<AD, B, D>
        andThen(Arrow<AD, C, D> k);
}
```

Listagem 24: Interface *Arrow*

A Listagem 25 declara a classe *Kleisli*. Esta implementa o contrato definido em *Arrow* e conta com os tipos genéricos *M*, *A* e *B*. Os argumentos e a implementação de *andThen* foram eliminados afim de manter a simplicidade na declaração do método.

Como *Kleisli* trabalha com tipos monádicos, seu primeiro parâmetro de tipo impõe uma ligação entre *M* e *Monad*. O tipo genérico *A* define a entrada da função encapsulada pelo construtor de classe e *B* o valor encapsulado pela mónade *M*. Além disso, os tipos genéricos declarados em *Kleisli* são repassados para *Arrow* e consequentemente formam os tipos de entrada e saída dos métodos sobrescritos.

```
class Kleisli<M extends Monad<B, ?>, A, B>
    implements Arrow<M, A, B> {
    ...

    public Kleisli(Function<A, M> f) {
        this.f = f;
    }

    public <D, MD extends Kind<D, ?>> Arrow<MD, A,
        D> andThen() {
        ...
    }
}
```

Listagem 25: Implementação concreta da interface *Arrow*

Para manter a consistência na implementação de *Kleisli*, o método *andThen* deve retornar uma instância de *Arrow* onde $M(D)$ extenda *Monad*. Contudo, *Arrow* impõe restrições através de *Kind* e estas devem ser mantidas na subclassificação. Portanto, o retorno de *andThen* definido por *Arrow* não pode ser expresso em *Kleisli*. Assim, a interface *Arrow* não é suficientemente genérica de modo que possa expressar as diferentes implementações de flechas.

D. Implementação de Kleisli

A implementação da classe *Kleisli* na Listagem 26 contem os tipos genéricos *M*, *B* e *C*. Como esta trabalha sob tipos monádicos e suas composições, *M* é declarado como sendo um *Monad* e representa o retorno da função encapsulada por *Kleisli*. O tipo *B* é a entrada da função e *C* o valor encapsulado no contexto de *M*, dado por $M(C)$.

Semelhante a *Monad*, valores são incluídos no contexto de *Kleisli* a partir construtor de classe. Este recebe uma função $B \rightarrow M$ que é armazenada e está disponível no escopo de classe. A aplicação da função encapsulada é disponibilizada pelo método *run*.

O operador de composição é expresso pelo método *andThen*. Apesar de conter diversas declarações de tipos genéricos na assinatura do método, *andThen* recebe uma instancia de *Kleisli* parametrizada por $M(D)$, *C* e *D*. Deste argumento dá-se a composição, onde o valor de *C* resultante da aplicação de $B \rightarrow M(C)$ é passada para $C \rightarrow M(D)$ de *k*. Deste encadeamento obtemos uma nova instancia de *Kleisli* definida por $M(D)$, *B* e *D*.

Com a implementação de *Kleisli* é possível compor mónades de forma semelhante a cadeia de *flatMap* e *map*. Na Listagem 27 *f*, *g* e *h* expressam funções na forma $A \rightarrow M(B)$, mais precisamente $Integer \rightarrow Maybe(Integer)$. O método *andThen* em conjunto com *run* consegue aplicar o valor passado como argumento na função encapsulada e repassa-lo para outras flechas, encadeando os valores entre as mónades.

```
Kleisli<Maybe<Integer>, Integer, Integer> f() {
    return new Kleisli<>(i -> Just.pure(i + 1));
}

Kleisli<Maybe<Integer>, Integer, Integer> g() {
    return new Kleisli<>(i -> Just.pure(i * 10));
}

Kleisli<Maybe<Integer>, Integer, Integer> h() {
    return new Kleisli<>(i -> Just.pure(i - 5));
}

h().andThen(f()).andThen(g()).run(5);
```

Listagem 27: Composição monádica com *andThen*

Kleisli mantém as regras das estruturas subjacentes. A aplicação de *h*, *f*, *g* e *run(5)* por *andThen* resulta em uma instância *Just(10)*. De maneira semelhante, se umas destas

```

class Kleisli<M extends Monad<?, ?>, B, C> {
    ...

    public Kleisli(Function<B, M> f) {
        this.f = f;
    }

    public M run(B b) {
        return f.apply(b);
    }

    public <D, MC extends Monad<C, MC>, MD extends Monad<D, ?>> Kleisli<MD, B, D> andThen(Kleisli<MD, C,
        D> k) {
        return new Kleisli<>(b -> (MD) ((MC) this.run(b)).flatMap(c -> (MC) k.run(c)));
    }
}

```

Listagem 26: Implementação de *Kleisli*

funções retornar *Nothing*, a cadeia não é executada e retorna este como resultado final.

Diversas leis devem ser satisfeitas por instâncias de *Arrow*. Todavia, como *Kleisli* não implementa uma interface genérica de *Arrow*, em sua implementação preocupou-se apenas com as leis de composição. Na Listagem 28 são implementados os axiomas de identidade e associatividade de *Kleisli*.

```

public void identity() {
    ... kf = new Kleisli<>(f);

    assertEquals(kf.run(a), f.apply(a));
}

public void associativity() {
    assertEquals(g.andThen(h).andThen(f).run(a),
        f.andThen(g.andThen(h)).run(a));
}

```

Listagem 28: Leis de *Kleisli*

c) *Identidade - identity*: a função f com assinatura na forma $A \rightarrow B$ é encapsulada por *Kleisli* e aplicada por $run(a)$. A mesma função f fora do contexto de *Kleisli* é aplicada por $apply(a)$. A integridade do axioma de identidade está na igualdade do valor obtido de ambas aplicações.

d) *Associatividade - associativity*: as funções f , g e h com assinatura na forma $A \rightarrow M(B)$ são encapsuladas em instâncias de *Kleisli* e compostas por *andThen* em diferentes ordens. A associatividade é garantida pela aplicação de ambas cadeias de funções por $run(a)$ e a igualdade de seus resultados.

VII. TRABALHOS RELACIONADOS

VIII. CONCLUSÃO

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,” Set. 2013, White Paper. [Online]. Available: <http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>
- [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,” Abr. 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] P. B. Menezes and C. E. Ramisch, “Características da teoria das categorias e sua importância para a ciência da computação,” in *STC 2006 - I Simpósio de Teoria das Categorias*. Universidade Federal do Rio Grande do Sul, Mar. 2006.

- [12] S. Awodey, *Category Theory - Oxford Logic Guides*, 2nd ed. Oxford University Press, 2010, vol. 52.
- [13] J. M. Hill and K. Clarke, "An introduction to category theory, category theory monads, and their relationship to functional programming," Technical Report QMW-DCS-681, Department of Computer Science, Queen Mary and Westfield College, Tech. Rep., 1994.
- [14] M. C. Pedicchio and W. Tholen, *Categorical Foundations Special Topics in Order, Topology, Algebra, and Sheaf Theory*. Cambridge University Press, 2004, no. 97.
- [15] S. L. Peyton Jones and P. Wadler, "Imperative functional programming," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1993, pp. 71–84.
- [16] P. Wadler, "Monads for functional programming," in *Advanced Functional Programming*. Springer, 1995, pp. 24–52.
- [17] —, "Comprehending monads," *Mathematical structures in computer science*, vol. 2, no. 04, pp. 461–493, 1992.
- [18] —, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1992, pp. 1–14.
- [19] 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.
- [20] B. O’Sullivan *et al.*, *Real world haskell: Code you can believe in*, 1st ed. O’Reilly Media, Inc., 2008.
- [21] B. Yorgey, "Typeclassopedia," <https://wiki.haskell.org/Typeclassopedia>, Mar. 2009, Acessado em 17/09/2015.
- [22] C. V. Hall *et al.*, "Type classes in haskell," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, pp. 109–138, 1996.
- [23] S. Marlow *et al.*, "Haskell 2010 language report," Disponível em: <http://www.haskell.org/onlinereport/haskell2010>, 2010.
- [24] J. Hughes, "Programming with arrows," in *Advanced Functional Programming*, ser. Lecture Notes in Computer Science, V. Vene and T. Uustalu, Eds. Springer Berlin Heidelberg, 2005, vol. 3622, pp. 73–129.
- [25] A. Courtney and C. Elliott, "Genuinely functional user interfaces," in *Haskell workshop*, 2001, pp. 41–69.
- [26] S. Lindley, P. Wadler and J. Yallop, "Idioms are oblivious, arrows are meticulous, monads are promiscuous," *Electronic Notes in Theoretical Computer Science*, vol. 229, no. 5, pp. 97–117, 2011.
- [27] J. Gosling and H. McGilton, "The java language environment," *Sun Microsystems Computer Company*, vol. 2550, 1995.
- [28] G. Bracha, "Generics in the java programming language," *Sun Microsystems Computer Company*, pp. 1–23, 2004.
- [29] G. Bracha *et al.*, "Making the future safe for the past: Adding genericity to the java programming language," *ACM Sigplan Notices*, vol. 33, no. 10, pp. 183–200, 1998.
- [30] B. C. Pierce, *Types and programming languages*. MIT Press, 2002.