Teoria das Categorias
Mónades e Flechas
em Java

Mozart L. Siqueira¹, Pablo M. Parada²
Ciência da Computação
Centro Universitário La Salle - Unilasalle
E-mail: mozarts@unilasalle.edu.br¹, pablo.paradabol@gmail.com²

***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 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, 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 caraterísticas 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 closures[1], a linguagem Java habilita a substituição de annonymous 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

[1]Lambda expressions ou closures são funções que não exigem vínculos de classe, como exemplo podendo ser atribuída a uma variável. Com esta caraterística, 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 permgem[2].

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

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[dr, "h = g \circ f"'] & B \arrow[d, "g"] \\ & C \end{tikzcd}$$

Figura 1: Funções entre coleções de objetos

Tal como a Teoria dos Grupos[3] 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 $f : A \to B$ e $g : B \to C$. 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 $id_a \in Hom_C(x, a)$;

(d) para cada três objetos $a, b, c \in Ob(C)$, uma função de composição $\circ : Hom_C(b, c) \times Hom_C(a, b) \to Hom_C(a, c)$;

Dado os objetos $a, b \in Ob(C)$, denota-se o morfismo $f \in Hom_C(a, b)$ por $f : a \to b$; 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 $f : a \to b$, tem-se $id_a \circ f = f = id_b \circ f$;

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

[3] Teoria 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 $f : a \to b$, $g : b \to c$ e $h : c \to d$. 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.

$$
\begin{tikzcd}
a \arrow[r, "id_a"] \arrow[d, "f"'] & a \arrow[d, "f"] \arrow[dl, "f"'] \\
b \arrow[r, "id_b"] & b
\end{tikzcd}
$$

Figura 2: Identidade

$$
\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "g \circ f"'] & b \arrow[d, "h \circ g"] \arrow[dl, "g"'] \\
c \arrow[r, "h"] & d
\end{tikzcd}
$$

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 $F : C \to D$ 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 $f : a \to b \in Hom_C(a, b)$ para um morfismo $F(f) : F(a) \to F(b) \in Hom_D(a, b)$.

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 $f : a \to b$ e $g : b \to c$. 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.

$C)$ $\qquad\qquad\qquad\qquad\qquad\qquad$ $D)$

$$
\begin{tikzcd}
a \arrow[loop left, "id_a"] \arrow[r, "f"] \arrow[dr, "g \circ f"'] & b \arrow[d, "g"] & & \mathrm{F(a)} \arrow[loop left, "F(id_a)"] \arrow[r, "f"] \arrow[dr, "F(g \circ f)"'] & \mathrm{F(b)} \arrow[d, "g"] \\
& c & F \Rightarrow & & \mathrm{F(c)}
\end{tikzcd}
$$

Figura 4: Funtor F

C. *Mónade – Endofuntor e Transformações Naturais*

Podendo ser visto como um padrão estrutural que ocorre diversos ramos da matemática [9], a construção fundamental mónade (também chamada de tripla) estrutura-se a partir de um endofuntor[4] e das transformações naturais[5] de identidade e multiplicação.

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

(a) um endofuntor T, tal que $T : C \to C$;

(b) nas transformações naturais:

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

(Multiplicação) $\mu : T^2 \to T$, 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.

$$
\begin{tikzcd}
T \arrow[r, "\eta_T"] \arrow[dr, "="'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T_\eta"'] \arrow[dl, "="] \\
& T &
\end{tikzcd}
$$

Figura 5: Identidade à Esquerda e à Direita

$$
\begin{tikzcd}
T^3 \arrow[r, "T_\mu"] \arrow[d, "\mu_T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}
$$

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;

[4] Um funtor que mapeia uma categoria para ela mesma.

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

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

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

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

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

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

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

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

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

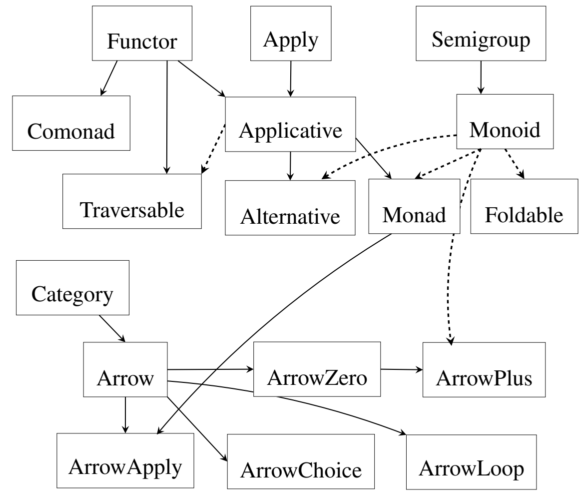

Figura 7: Hierarquia de type classes[6]

```
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 $Int \to Int \to Bool$ e $Char \to Char \to Bool$ 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 $False$.

A. Monad

Para a programação funcional, mónades oferecem um contexto computacional que encapsula efeitos colaterais. Além disso, tal contexto monádico permite o encadeamento de

[6]Fonte: https://wiki.haskell.org/Typeclassopedia. Acessado em 17/09/2015.

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

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 $a \rightarrow m\ b$ 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 Then[8]

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 Notation[8]

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/S[12][10]

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 $>>>$.

```
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 $Arrow$ de f [25].

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

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

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

[11]Um 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.

A segunda operação é definida simbolicamente por $(>>>)$ com assinatura $a\ b\ c \to a\ c\ d \to a\ b\ d$. 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, denotada simbolicamente por $(>>>)$, utiliza-se $(.)$ e $flip$ [24]. Assim, define-se o tipo $Arrow$ $(\to)$, capaz de denotar funções ordinárias. Sua assinatura é dada por $a \to b$, sendo a e b tipos quaisquer.

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

Listagem 12: Arrow - Composição de Funções[12]

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, $Arrow\ (Kleisli\ m)$ generalizam mónades[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. Mónades em Java

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 *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.

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

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

[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.