Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 25, Num. 3 (2018) 75-88

RESEARCH ARTICLE

Haskell Type System Analysis

Análise do Sistema de Tipos de Haskell

Rafael Castro G. Silva[1], Karina Girardi Roggia[1], Cristiano Damiani Vasconcellos[1]

Abstract: Types systems of programming languages are becoming more and more sophisticated and, in some cases, they are based on concepts from Logic, Type Theory and Category Theory. Haskell is a language with a modern type system and it is often singled out as an example using such theories. This work presents a small formalization of the Haskell type system and an analysis based on the mentioned theories, including its relation with the Intuitionist Propositional Second Order Logic and its logical characteristics, if there is a category in its type system and how monads are just monoids in the category of Haskell's endofunctors.

Keywords: Haskell — Categories — Logic — Types

Resumo: Sistemas de tipos de linguagens de programação estão cada vez mais sofisticados e, em alguns casos, são baseados em conceitos da Lógica, da Teoria dos Tipos e da Teoria das Categorias. Haskell é uma linguagem com um sistema de tipos moderno e frequentemente é apontado como um exemplo que utiliza tais teorias. Este trabalho apresenta uma pequena formalização do sistema de tipos de Haskell e uma análise com base nas teorias mencionadas, incluindo a sua relação com a Lógica Proposicional de Segunda Ordem e suas características lógicas, se há uma categoria em seu sistemas de tipos e como mônadas são apenas monoides na categoria dos endofuntores de Haskell.

Palavras-Chave: Haskell — Categorias — Lógica — Tipos

[1] *Departamento de Ciência da Computação - Universidade do Estado de Santa Catarina (UDESC) - Joinville - SC - Brasil*
Corresponding author: rafaelcgs10@gmail.com, karina.roggia@udesc.br, cristiano.vasconcellos@udesc.br
DOI: https://doi.org/10.22456/2175-2745.82395 • **Received:** 29/04/2018 • **Accepted:** 16/08/2018

1. Introdução

O sistema de tipos de Haskell é decorrência de anos de pesquisa em Teoria dos Tipos. Suas principais características são segurança, tipos de dados algébricos (*Algebraic Data Types* ou ADT), e polimorfismo paramétrico e *ad-hoc*. Haskell estende o sistema de tipos Damas-Milner[1], principalmente, com o suporte de sobrecarga. A principal característica desse sistema é a inferência de tipos e o suporte ao polimorfismo paramétrico, sendo definido um algoritmo que sempre encontra o tipo principal de uma expressão para qual exista uma prova construtiva.

Propositions as Types é uma relação entre provas na lógica construtivista e provas em sistemas de tipos [2], que é principalmente conhecida como Isomorfismo de Curry-Howard. Esta relação estipula que proposições são tipos e provas construtivas são termos do Cálculo Lambda Tipado. A equivalência relaciona diretamente o Cálculo Lambda Simplesmente Tipado com a Lógica Intuicionista na Dedução Natural de Gentzen [3]. Joachim Lambek, posteriormente, descobriu que tal correspondência também está presente dentro das Categorias Cartesianas Fechadas, o que resulta na tripla correspondência Curry-Howard-Lambek [4].

Não somente Haskell, mas toda a Teoria das Linguagens de Programação colheu diversos frutos dessa tripla relação, pois explica de três maneiras diferentes conceitos fundamentais da programação, como operadores lógicos, tipos de dados, funções e computações. Interpretar um mesmo conceito de diversas maneiras possibilita utilizar resultados prévios distintos, pois frequentemente uma teoria matemática é complementar à outra.

Muitos conceitos de Curry-Howard-Lambek foram aplicados na especificação de linguagens de programação. A ideia de quantificação universal nasceu na lógica e inspirou o polimorfismo universal, o qual tem se mostrado ser uma funcionalidade com bastante aceitação, visto que mesmo linguagens de programação imperativas como Java e C++ adotaram-no. A Mônada é um conceito da teoria das categorias, a qual inspirou uma maneira de tratar IO em Haskell e, posteriormente, também foi adotado por outras linguagens de programação como, por exemplo, Scala.

Este trabalho propõe uma análise matematicamente coerente quanto às alegações geralmente feitas sobre Haskell e o seu sistema de tipos. Conceitos da lógica e Teoria das Categorias são explicados por meio de exemplos em Haskell, logo o leitor familiarizado com a linguagem pode entendê-los mais facilmente. A principal contribuição deste trabalho é uma análise rigorosa quanto aos aspectos formais mencionados e verificar quais são as implicações práticas disso. Assim, busca-se identificar características positivas e falhas nas analogias com as teorias matemáticas. A segunda seção deste trabalho trata sobre questões relacionadas a lógica em Haskell e a terceira sobre uma possível categoria cartesiana fechada no sistema de tipos de Haskell.

2. Sistema de Tipos de Haskell

Como qualquer linguagem funcional, Haskell é baseada em Cálculo Lambda, mas para ser útil como uma linguagem de programação a sua sintaxe é consideravelmente enriquecida para considerar constantes, operações, casamento de padrões, definições *let*, declarações de tipos de dados algébricos, classes de tipo, etc. Seria inviável analisar toda a linguagem Haskell, por essa razão esta seção utiliza uma simplicaficação da sintaxe e das regras de inferência de Haskell. A principal diferença do sistema apresentado aqui em relação ao Damas-Milner [1] são as classes de tipo, que servem para introduzir a sobrecarga de operadores.

O polimorfismo do Sistema Damas-Milner é conhecido como paramétrico, ou universal, e está presente em linguagens como *Standard* ML (ML), Miranda e Haskell. As duas primeiras precedem Haskell e não contam com classes de tipo, que é a forma utilizada em Haskell para tratar sobrecarga de funções (*function overloading*). A sobrecarga de funções ocorre quando é possível fornecer mais que uma implementação para uma mesma função, essas implementações diferem nos tipos dos argumentos e, possivelmente, no tipo do retorno. O polimorfismo *ad-hoc*, também denominado de polimorfismo com restrições, é caracterizado pelo uso de funções sobrecarregadas em expressões. Essas expressões são válidas somente para um grupo específico de tipos, o grupo que possui uma definição sobrecarregada da função.

A definição da sintaxe de Haskell é dada no Haskell Report [5], na Figura 1 é apresentado o núcleo da linguagem ML estendido com Classes de Tipos para o suporte a sobrecarga. As regras de inferência para esta linguagem, denominada **AHDM**, são apresentadas na Figura 2. Essa formalização é baseada no trabalho de Wadler e Blott [6], que introduz classes de tipos em Haskell.

As declarações `over` e `inst` representam, respectivamente, a declaração de identificadores sobrecarregados e as suas instâncias. Diferente das demais expressões, `over` e `inst` precisam de explícitas assinaturas de tipos. O construtor de tipo χ e o construtor de valor ϑ são tokens que começam com letra maiúscula e servem, respectivamente, como identificador para novos tipos (compostos) e valores. Um tipo conveniente de ser criado é o tipo tupla (*Pair*), de maneira que `fst` em:

```
data Pair (α0 α1) = Pair (x0 x1) in
let fst = λPair (x0 x1).x0 in
fst (Pair (1 2))
```

Na realidade, a regra de produção `data` generaliza pares e viabiliza definir tuplas de qualquer aridade.

O tipo $(x::\tau).\rho$ é referido como tipo predicativo e $(x::\tau)$ como predicado. Predicados podem ser interpretados como o predicado x é uma instância de τ.

A formalização original de classes de tipos, apresentada em [6], tem regras para derivar fórmulas de formato:

$$A \vdash e :: \sigma/\bar{e},$$

que podem ser lidas como, "no contexto A, a expressão e tem o tipo σ e a conversão para $\bar{e}$". Cada regra de derivação acompanha uma conversão, então as derivações são pares tipo/conversão. A parte da conversão $(/\bar{e})$ pode ser omitida e mantidos apenas os tipos derivados, mas o contrário não é possível, pois conversões dependem dos tipos. Por simplicidade, opta-se aqui pela omissão das conversões. Adiciona-se, ainda, uma regra para deduzir tipo de valores constantes.

Como usual, por simplicidade, foi omitido a regra *letrec*, uma vez que é possível declarar funções recursivas assumindo a existência do operador de ponto fixo, sem que isso tenha impacto no sistema de tipos. O operador de ponto fixo respeita a seguinte equivalência: `fix :: (a -> a) -> a`, o qual respeita a equivalência

```
fix  f ≡ f (fix f)
```

A função `seq :: a -> b -> b` de Haskell também não pode ser definida a partir das demais regras e serve para introduzir avaliação estrita na linguagem. De certa maneira, `seq` magicamente avalia de maneira estrita o seu primeiro argumento e retorna o segundo. "Magicamente" porque já que sua implementação é feita pela introdução de uma dependência virtual de valores entre o resultado e o primeiro argumento. Defini-se `seq` conforme a relação

```
seq ⊥ b ≡  ⊥
seq _ b ≡ b
```

sendo que $\perp$ representa uma recursão que não chega a forma normal. Considera-se que neste sistema `fix` e `seq` estão disponíveis no contexto sempre que necessários.

Os Tipos de Dados Algébricos são uma das principais características deste sistema, pois fornecem uma maneira flexível de definir novos tipos de dados. A sua formalização é dada pelas regras *data*, *case* e *dataSpec*, os quais respectivamente servem para definir novos dados, desconstruir dados e especializar os tipos dos dados. A regra *data* é definida com o auxilio das duas definições recursivas abaixo.

$$K_i(0) = \chi\,\alpha_1 \ldots \alpha_n$$
$$K_i(j) = \tau_{i,j} \to K_i(j-1)$$

Variáveis	x
Constantes	c
Expressões	e ::= x | e$_0$ e$_1$ | λx.e | **let** x **=** e$_0$ **in** e$_1$ | **case** e **of** { p$_0$ **=>** e$_0$ | ... | p$_m$ **=>** e$_m$ } | **over** x :: σ **in** e | **inst** x :: σ **=** e$_0$ **in** e$_1$ | **data** $\chi\alpha_0...\alpha_n$ **=** $\vartheta_0\tau_0...\tau_{k(0)}$ | ... | $\vartheta_m\tau_0...\tau_{k(m)}$ **in** e$_1$
Padrões	p ::= x | c | ϑ p$_0$ ... p$_m$
Construtor de dado	ϑ
Variáveis de tipo	α
Tipos de constantes	B$_c$
Construtor de tipo	χ
Monotipos	τ ::= $(\tau \to \tau')$ | χ | α | τ τ'
Tipos predicativos	ρ ::= $(x::\tau).\rho$ | τ
Politipos	σ ::= $\forall\alpha.\sigma$ | ρ

Figura 1. Sintaxe de termos e tipos em **AHDM**.

$$C_i(0) = \vartheta_i$$
$$C_i(j) = (C(j-1)\,\tau_{i,j})$$

Uma substituição de tipo $\mathbb{S}$ em um tipo é uma sequência:

$$[\tau_1/a_1, \tau_2/a_2, \tau_3/a_3, ..., \tau_n/a_n],$$

sendo τ_i são quaisquer monotipos e a_i variáveis de tipo. A aplicação de $\mathbb{S}$ é definida recursivamente por:

$$\begin{aligned}
\mathbb{S}(a_i) &\equiv \tau_i \\
\mathbb{S}(B_c) &\equiv B_c \\
\mathbb{S}(\tau \to \tau') &\equiv \mathbb{S}(\tau) \to \mathbb{S}(\tau') \\
\mathbb{S}((x :: \tau).\rho') &\equiv (x :: \mathbb{S}(\tau)).\mathbb{S}(\rho') \\
\mathbb{S}(\forall\alpha.\sigma) &\equiv \forall\mathbb{S}(\alpha).\mathbb{S}(\sigma).
\end{aligned}$$

O conjunto de todas as variáveis de tipo livres de τ, denotado por $FTV(\tau)$, é definido recursivamente por:

$$\begin{aligned}
&FTV(\alpha) = \{\alpha\} \\
&FTV(\chi) = \emptyset \\
&FTV((x :: \tau).\rho) = FTV(\tau) \cup FTV(\rho) \\
&FTV(\tau \to \tau') = FTV(\tau) \cup FTV(\tau') \\
&FTV(\tau\tau') = FTV(\tau) \cup FTV(\tau') \\
&FTV(\forall\alpha.\sigma) = FTV(\sigma) \setminus \{\alpha\}
\end{aligned}$$

O contexto relaciona identificadores com tipos de três maneiras diferentes:

1. $(x ::_o \sigma)$ para identificadores sobrecarregados;
2. $(x ::_i \sigma)$ para instâncias declaradas de identificadores sobrecarregados;
3. $(x :: \sigma)$ para variáveis ;

Nota 2.1. $Eq\,\alpha$ é uma abreviação para $\alpha \to \alpha \to Bool$.

A condição de instanciação/especialização de um tipo em AHDM é uma extensão de Damas-Milner, que utiliza o contexto para determinar se uma instância é correta. Por exemplo,

$$\forall a.(eq :: Eq\,a).a \to a >_A Int \to Int$$

é correto se existe uma declaração de *eq* para o tipo *Int* num dado contexto A, ou seja, $(eq ::_i Eq\,Int) \in A$. Assim como em Damas-Milner, um politipo σ é uma especialização de outro politipo σ' se, e somente se, a condição abaixo for satisfeita.

$$\frac{\rho' \equiv [\sigma_i/\alpha_i]\rho \qquad \beta_i \notin FTV(\forall\alpha_1...\alpha_n.\rho)}{\forall\alpha_1...\alpha_n.\rho > \forall\beta_1...\beta_m.\rho'}$$

Além de politipos também há tipos predicativos em AHDM, então é necessário uma outra condição. No contexto A, o tipo predicativo ρ' é uma especialização de ρ (relação denotada por $\rho >_A \rho'$), se e somente se, ambos tem o mesmo monotipo e se para cada predicado $(x :: \tau)$ em ρ, ou

1. o mesmo predicado $(x :: \tau)$ está em ρ', ou
2. o predicado pode ser eliminado no contexto A.

Um predicado $(x :: \tau)$ pode ser eliminado no contexto A se, e somente se, ou:

1. $(x :: \tau) \in A$, ou
2. $(x ::_i \sigma') \in A$ e $\sigma' >_A \tau$.

$$\frac{}{x :: \tau \vdash x :: \tau}\ (var) \qquad \frac{}{x ::_i \tau \vdash x :: \tau}\ (var-inst) \qquad \frac{}{\vdash c :: B_c}\ (const)$$

$$\frac{A, \vartheta_i :: \forall\alpha_1 ... \alpha_m . K_i(n_i) \vdash e :: \tau \qquad \text{para } i = 1...h,\ FTV(\tau_{i,j}) \subseteq \{\alpha_1, ..., \alpha_m\}}{A \vdash (\textbf{data}\, \chi\, \alpha_1 ... \alpha_m = C_1(n_1) | ... | C_h(n_h)\, \textbf{in}\, e) :: \tau}\ (data)$$

$$\frac{A \vdash e :: \tau' \qquad A, p_i :: \tau' \vdash e_i :: \tau \qquad \text{para } i = 0..m}{A \vdash (\textbf{case}\, e\, \textbf{of}\, \{p_0 => e_0 \mid ... \mid p_m => e_m\}) :: \tau}\ (case)$$

$$\frac{\sigma < \tau}{A, \vartheta :: \sigma \vdash \vartheta :: \tau}\ (dataSpec)$$

$$\frac{A \vdash e :: \tau' \to \tau \qquad A \vdash e' :: \tau'}{A \vdash (ee') :: \tau}\ (app) \qquad \frac{A \vdash e :: \tau}{A \setminus x : \tau' \vdash (\lambda x.e) :: \tau' \to \tau}\ (abs)$$

$$\frac{A \vdash e :: \sigma \qquad A, x :: \sigma \vdash e' :: \tau}{A \vdash (\textbf{let}\, x = e\, \textbf{in}\, e') :: \tau}\ (let)$$

$$\frac{A \vdash e :: \sigma \qquad \alpha \notin FTV(A)}{A \vdash e :: \forall\alpha.\sigma}\ (gen) \qquad \frac{A \vdash e :: \sigma \qquad \sigma < \sigma'}{A \vdash e :: \sigma'}\ (spec)$$

$$\frac{A, x ::_o \sigma \vdash e :: \tau}{A \vdash (\textbf{over}\, x :: \sigma\, \textbf{in}\, e) :: \tau}\ (over)$$

$$\frac{A, x ::_i \sigma' \vdash e' :: \sigma' \qquad A, x ::_i \sigma' \vdash e :: \tau'}{A \vdash (\textbf{inst}\, x :: \sigma' = e'\, \textbf{in}\, e) :: \tau}\ (inst)$$

$$\frac{A, x :: \tau \vdash e :: \rho}{A \vdash e :: (x :: \tau).\rho}\ (pred) \qquad \frac{A \vdash e :: (x :: \tau).\rho \qquad A \vdash x :: \tau}{A \vdash e :: \rho}\ (rel)$$

Figura 2. Regras de inferência de tipos do sistema **AHDM**.

Como exemplo tome o contexto $\{eq ::_i Eq\, Int\}$ e $\sigma \equiv \forall a.(eq :: Eq\, a).a \to a$. Pela condição de especialização de politipo tem-se:

$$\forall a.(eq :: Eq\, a).a \to a > (eq :: Eq\, Int).Int \to Int$$

O predicado $(eq :: Eq\, Int)$ pode ser eliminado, pois há no contexto uma declaração de instância de *eq* cujo tipo pode ser especializado para $(Eq\, Int)$. Portanto,

$$(eq :: Eq\, Int).Int \to Int >_A Int \to Int.$$

Observe que ambos atendem a condição de terem o mesmo monotipo.

Nota 2.2. As regras $(pred)$, (rel) e $(inst)$ tem como condição: $x ::_o \sigma \in A$.

2.1 Representatividade de AHDM no Haskell

O sistema definido acima não representa completamente a linguagem Haskell, tão pouco as suas variadas extensões. No entanto, o objetivo de **AHDM** é representar as principais funcionalidades do sistema de tipos de Haskell, que são polimorfismo paramétrico e *ad-hoc*, funções de ordem superior, recursão monomórfica, tipos de dados algébricos, tipos recursivos/mutuamente recursivos e tipos aninhados.

Um questionamento relevante é o que do sistema de tipos de Haskell não foi considerado em *AHDM*. Ignorou-se funcionalidades que não foram consideradas relevantes para a análise das próximas seções ou cujas formalizações seriam complexas demais. Com base nesses critérios, optou-se por não considerar em **AHDM** as seguintes funcionalidades do sistema de tipo de Haskell:

- Recursão mútua: Haskell trata a inferência de declarações mutualmente recursivas adotando uma ordenação no processo de inferência, em alguns casos ocasionando que a função tenha um tipo menos geral do que o possível. Por exemplo, considerando um contexto onde a função `const` tem tipo `a - >b -> a` o tipo das funções

```
f x = const x g
g x = f 'A',
```

é inferido como se a declaração fosse:

```
f x = let g y = f 'A' in const x g.
```

Dessa forma, a função `f` que poderia ter tipo polimórfico (`a -> a`) é tratada como monomórfica por ocorrer no escopo da sua própria definição, sendo inferido o tipo **`Char -> Char`**.

- Tipos de dados abstratos: A abstração de dados em Haskell geralmente é feita pela definição de um **`module`**, a qual seria possível formalizar como uma característica do sistema de tipos, porém considerá-la como tal iria adicionar ainda mais complexidade ao sistema e isso não caracteriza uma funcionalidade peculiar de Haskell.

- Tipos de ordem superior (*kinds*): Quando especificando uma classe de tipo em Haskell pode-se assinar o parâmetro com um *kind* (tipo do tipo), porém a aplicação de tipos deixa implícito a ideia de que certos tipos recebem tipos como argumentos.

- Recursão polimórfica: O operador de ponto fixo tem o tipo $\forall a.(a \to a) \to a$ e uma vez que a variável de tipo a é especializada, então todas as chamas recursivas devem utilizar essa mesma especialização, ou seja, a recursão é monomórfica. A inferência de tipos de expressões que apresentam recursão polimórfica é um problema indecidível, pois se reduz ao problema da semi-unificação[7]. Haskell, porém, permite que programas com recursão polimórfica sejam escritos se os tipos forem assinados.

- Abstração polimórfica: programas assinados em Haskell podem receber funções e utilizá-las de maneira polimórfica, por exemplo

```
foo :: (forall a. a -> a) ->
 ↪  (Char,Bool)
foo f = (f 'c', f True).
```

  Porém, assim como a recursão polimórfica, a sua inferência é indecidível.

As análises das próximas seções utilizam a sintaxe de Haskell por ser de comum entendimento, porém qualquer exemplo de código pode ser traduzido para **AHDM**. Em especial, o termo `undefined :: a` pode ser representado em **AHDM** por

```
let undefined = fix (λx.x) in
 ↪  undefined.
```

3. Triplo Isomorfismo Curry-Howard-Lambek na Teoria das Linguagens de Programação

A correspondência Curry-Howard-Lambek é uma tripla relação entre Teoria dos Tipos, Teoria da Prova e Teoria das Categorias. Inicialmente formulado como uma correspondência entre tipos e proposições (*Propositions as Types*), foi posteriormente estendida por Lambek por meio de uma interpretação categórica [8]. Nesta seção é discutido alguns trabalhos em linguagens de programação que utilizam essas ideias.

O Isomorfismo de Curry-Howard foi descoberto, em 1969, por William Alvin Howard, com base nos resultados prévios de Haskell Curry e relaciona diretamente o Cálculo Lambda Simplesmente Tipado com o fragmento intuicionista da Dedução Natural de Gentzen [3]. A essência de Curry-Howard é que proposições lógicas são sentenças que podem ser tautológicas da mesma forma que tipos podem corresponder a assinaturas de programas. Por esse motivo, outro nome comum para essa ideia é *Propositions as Types*. Segundo Philip Wadler [2], isso é uma noção com profundidade: vai na raiz das linguagens de programação funcional, explica funções, *records*, variantes, polimorfismo paramétrico, tipos de dados abstratos e continuações.

A ideia de utilizar tipos como proposições lógicas possibilita assertar propriedades de programas através da sua assinatura de tipos. Por exemplo, uma assinatura de tipos pode expressar que a entrada de um programa é uma lista de números e a saída é essa mesma lista de números, porém ordenada. Especificações mais fortes como essa são frutos da Teoria de Tipos Dependentes, que além de possibilitar a codificação de proposições lógicas de ordem superior, com quantificação universal e existencial sobre proposições, também possibilitam que tipos dependam de valores. Essa combinação de ordem superior e tipos dependentes fornecem uma forma de especificar a corretude de programas. Em linguagens com tipos dependentes, como Agda, ATS, Coq, Idris e F*, o verificador de tipos (*type checker*), que faz parte do compilador, assegura que o programa implementado habita (ou não) a assinatura de tipos especificada.

A Teoria das Categorias é uma teoria sobre funções (morfismos) assim como Teoria dos Tipos e dado o seu alto grau de abstração não é de se surpreender que exista uma relação entre ambas. De maneira simplificada, tipos são vistos como objetos e funções como morfismos. O que pode implicar na existência de categorias em sistemas de tipos de linguagens de programação. Além disso, Teoria das Categorias é uma forma de estudar a semântica das linguagens de programação de maneira a ignorar questões sintáticas, pois um dos seus princípios é a composição de morfismos.

Um dos primeiros a perceber isso foi Eugenio Moggi com a sua Semântica Categórica de Computações [9] por meio da estrutura categórica mônada, que permitem a modelagem da semântica de uma linguagem de programação com computações parciais. Isso resultou no conceito *Computation as Monads*.

Mônadas foram criadas na Teoria das Categorias para expressar certas estruturas algébricas superiores, posteriormente cientistas da computação perceberam que mônadas não só servem para modelar a semântica denotacional de linguagens de programação, mas também para generalizar compreensão de lista e estruturar linguagens de programação puras[10], mais especificamente para generalizar continuações (*Continuation-passing style*) e integrar computações com efeitos

colaterais, como *input-ouput*, não-determinismo, exceções, paralelismo/concorrência e estado [11].

Em resumo, a descoberta dessa tripla relação resultou numa boa fundação para o desenvolvimento das linguagens de programação funcionais, que são baseadas em Cálculo Lambda, em especial aquelas que tem polimorfismo paramétrico e com tipos dependentes.

4. Análise Lógica do Sistema de Tipos de Haskell

Gerhard Gentzen por volta de 1930 criou um sistema de prova conhecido como Dedução Natural [12], que utiliza regras formais similares às apresentadas na Seção 2. É possível representar diversas lógicas e provas em Dedução Natural, em especial a lógica clássica e a lógica intuicionista.

Uma lógica tem uma sintaxe, uma semântica e alguns postulados ou princípios. Por exemplo, a lógica clássica tem uma sintaxe para suas proposições, uma semântica dada por tabela verdade e o princípio do terceiro excluído, portanto tem-se o termo $A \vee \neg A$ como uma tautologia mesmo se não há provas sobre A ou $\neg A$.

A lógica intuicionista é uma lógica construtivista, ou seja, rejeita provas que não apresentem a construção do objeto a ser provado. Em suma, provas por contradição não são aceitas. O intuicionismo foi motivado pela noção de efetivamente computável, que é uma consequência direta das descobertas de Kurt Gödel, Alan Turing e Alonzo Church sobre indecibilidade de sistemas.

Na lógica clássica, a semântica é baseada nas tabelas verdades, já na lógica intuicionista a semântica é dada pela interpretação de Brouwer-Heyting-Kolmogorov (BHK). Cada fórmula determina um processo de construção de um objeto. Assim:

- uma construção de $A \wedge B$ é um par de construções, que indica a existência de uma construção de A e uma construção de B;
- $A \vee B$ é uma construção de A ou uma construção de B;
- $A \to B$ é um procedimento tal que dadas construções de A, obtém-se construções de B;
- $\forall x.\phi(x)$ é um procedimento que converte todo x numa prova de $\phi(x)$;
- nada é uma construção de $\bot$. Assim, $\neg A$ é definido como $A \to \bot$, o que significa que $\neg A$ é um procedimento que gera a constante lógica sempre falso ($\bot$). Uma vez que $\bot$ não pode ser construído[1], então $\neg A$ também não pode.

No intuicionismo, portanto, se uma proposição é verdadeira deve ser possível demostrá-la por meio de uma prova construtivista, ou seja, apresentando uma instância.

[1]Prova presente no livro Teoria das Categorias para Ciência da Computação [13].

O princípio construtivista encaixa-se perfeitamente com o sistema de tipos de Haskell. Não é possível afirmar que um tipo é habitado se não for fornecido um programa com esse tipo. Por exemplo, a função

```
const :: a -> b -> a
const x _ = x
```

é uma função que recebe dois argumentos e retorna somente o primeiro, portanto representa exatamente o procedimento esperado pela proposição $\tau \to \sigma \to \tau$. Basta interpretar argumentos como suposições e o retorno como a conclusão. No entanto, nota-se que Haskell tem o quantificador $\forall$ implícito em seus tipos polimórficos, portanto a interpretação mais precisa seria considerar que `const` converte todos os argumentos de tipos `a` e `b` numa prova de `a -> b -> a`.

O leitor atento deve ter observado que a interpretação BHK do quantificador $\forall$ se refere a variáveis atuando em predicados. O Isomorfismo de Curry-Howard trata de uma lógica proposicional e não de uma lógica predicativa. De fato, a lógica que deseja-se é a Lógica Proposicional de Segunda Ordem, a qual é obtida ao adicionar funcionalidades de segunda ordem na lógica proposicional [14]. Essa lógica também pode representar a semântica clássica e intuicionista, e têm uma Dedução Natural, cujas regras para introduzir e eliminar quantificadores $\forall$ são iguais do sistema Damas-Milner (ver Figura 3).

$$(p \notin FTV(\Gamma))\ \frac{\Gamma \vdash \sigma}{\Gamma \vdash \forall p.\sigma}\ (\forall_I) \qquad \frac{\Gamma \vdash \forall p.\sigma}{\Gamma \vdash [\tau/p]\sigma}\ (\forall_E)$$

Figura 3. Regras de introdução e eliminação de quantificadores da Segunda Ordem Proposicional.

Os tipos quantificados de Haskell são os do sistema Damas-Milner, conhecidos como *typeschemes*. Outro sistema de tipos com quantificação é o Sistema-F, também conhecido como Cálculo Lambda Polimórfico, que foi descoberto duas vezes: uma pelo lógico Jean-Yves Girard [15] e pelo cientista da computação John Reynolds [16]. Damas-Milner é um fragmento do Sistema-F, no sentido de que todo programa tipável em Damas-Milner também é no Sistema-F. A gramática de tipos do Sistema-F permite que quantificadores aconteçam de maneira mais flexível:

$$\sigma ::= \ldots \mid \sigma \to \sigma' \mid \forall \alpha.\sigma.$$

O Sistema-F, pelo Isomorfismo de Curry-Howard, corresponde a um fragmento da Lógica Proposicional de Segunda Ordem Intuicionista (LPSOI). Ou seja, tem-se $\Gamma \vdash M : \sigma$ no Sistema-F se, e somente se, tem-se $\Gamma \vdash \sigma$ no fragmento $\{\forall, \to\}$ de LPSOI. Consequentemente, o sistema Damas-Milner é um fragmento ainda menor de LPSOI.

O sistema **AHDM** é uma extensão do Damas-Milner, pois adiciona conectivos lógicos além da implicação e também possibilita uma forma de polimorfismo restrito (*ad-hoc*). As

regras dos Tipos de Dados Algébricos de **AHDM** estão relacionadas com as regras de disjunção e conjunção da LPSOI (ver Figura 4). A regra *data* corresponde simultaneamente as regras $(\wedge_I)$ e $(\vee_I)$. E a regra *case* combinada com a regra *dataSpec* representa as regras $(\wedge_I)$ e $(\vee_I)$ quando as suas proposições não são quantificadas.

$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \sigma'}{\Gamma 1 \vdash \sigma \wedge \sigma'}\,(\wedge_I)$$

$$\frac{\Gamma \vdash \sigma \wedge \sigma'}{\Gamma \vdash \sigma}\,(\wedge_{E_1}) \qquad \frac{\Gamma \vdash \sigma \wedge \sigma'}{\Gamma \vdash \sigma}\,(\wedge_{E_2})$$

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma \vee \sigma'}\,(\vee_{I_1}) \qquad \frac{\Gamma \vdash \sigma'}{\Gamma \vdash \sigma \vee \sigma'}\,(\vee_{I_2})$$

$$\frac{\Gamma, \sigma' \vdash \sigma \qquad \Gamma, \sigma'' \vdash \sigma \qquad \Gamma, \vdash \sigma' \vee \sigma''}{\Gamma \vdash \sigma}\,(\vee_E)$$

Figura 4. Regras de introdução e eliminação da conjunção e disjunção da Segunda Ordem Proposicional.

As regras $(over)$, $(inst)$, $(pred)$ e (rel) servem para introduzir uma forma de polimorfismo com restrições. A restrição é um identificador parametrizado por uma variável de tipo e que garante a existência de uma implementação uma função (ou mais) que utiliza essa variável de tipo. Por exemplo, uma função de tipo $\forall a.\, Num\, a.a \to a$ garante a implementação do operador de arimética $+$ com uma instância do tipo $\forall a.a \to a \to a$. Isso, via Curry-Howard, pode ser visto como uma proposição $\forall p.p \to p$, tal que há uma prova para esse p de uma instância da propriedade $\forall p.p \to p \to p$. Não há como impor a existência dessa propriedade pelas regras de LPSOI, porém é possível interpretá-las como proposições quantificadas que respeitam uma determinada propriedade.

4.1 Características da Lógica de AHDM

O termo `undefined :: a` serve para representar uma computação com erro ou um loop infinito, sua avaliação gera uma exceção. Uma possível implementação do `undefined` é dado pelo seguinte loop infinito:

```
undefined :: a
undefined = undefined.
```

Na realidade, qualquer loop infinito tem a mesma semântica do `undefined` (não terminação) e é apenas uma escolha de design definir `undefined` como uma exceção. Inversamente, qualquer exceção também significa não terminação[2]. Apesar de não existir especificação formal para semântica operacional de Haskell, qualquer não terminação é considerada como um valor *bottom* ($\perp$).

Uma vez que `undefined` é definido com o tipo polimórfico `a`, então é possível criar qualquer outro tipo sintaticamente correto por meio de substituições de tipo. Portanto, `undefined` tem qualquer tipo e o inverso também é verdade: qualquer tipo é habitado por ele.

[2]Exceções também dão alguma informação do que deu errado.

O Isomorfismo de Curry-Howard além de relacionar tipos com proposições e termos com provas, também relaciona tipos habitados com teoremas. Como todos os tipos do sistema de tipos de Haskell são habitados pelo termo `undefined` (valor *bottom*), então a respectiva lógica isomorfa tem todas as proposições como teoremas e, assim, poder-se-á concluir que o sistema de tipos de Haskell é inconsistente.

No entanto, afirmar que Haskell é inconsistente pode ser uma imprecisão, pois segundo [17]:

> Diz-se que uma teoria dedutiva é consistente se não possuir teoremas contraditórios, um dos quais, a negação do outro. Caso contrário, a teoria diz-se inconsistente (ou contraditória). Uma teoria chama-se trivial se todas as fórmulas (ou sentenças) de sua linguagem forem nela demonstráveis; em hipótese contrária, diz-se não-trivial.

Uma relação entre trivialidade e inconsistência foi usada na afirmação de que Haskell é inconsistente: se um sistema é trivial, então é inconsistente pois tanto $\neg p$ e p são teoremas. O problema é que Haskell tem um sistema de tipos que projeta uma lógica intuicionista positiva, ou seja, não há negação $(\neg)$ e nem sempre falso $(\perp)$[3]. Portanto, seria mais preciso afirmar que Haskell é trivial (todos os tipos são provados pelo `undefined`) e consistente, pois não é possível formar um tipo que seja a contradição de outro. Alguém, ingenuamente, poderia afirmar que Haskell tem o tipo falso **`data Void`** e, também, a função `absurd :: Void -> a` garante a trivialidade, pois do falso tudo se segue. No entanto, **`Void`** não é realmente o tipo $\perp$, pois na interpretação BHK nada deveria ser uma prova de falso, o que não é o caso, pois `undefined` prova tudo.

É relevante como isso afeta a segurança da tipagem. Por exemplo, se o tipo `a -> b` é habitado, então não há garantia que não seja habitado pela função `unsafeCoerce :: a -> b`[4], que deixa de lado a verificação de tipo e permite *casting* arbitrário de tipos. Se o sistema fosse não-trivial, essa questão nem poderia ser levantada.

O esperado argumento favorável ao sistema de tipos de Haskell é que apesar de trivial, o mesmo garanta que não há como somar uma **`String`** com um **`Int`** ou inserir um **`Bool`** numa lista de **`Char`**. Na prática, o que um sistema de tipos de uma linguagem de programação precisa para ser útil é garantir que o programador não cometa erros na definição e na composição de funções, que poderiam resultar em um programa sem sentido semântico. Essa garantia é conhecida como *Type Safety*, que é definida por Pierce[18] como *Type Safety* = *Progress* + *Preservation*, sendo

1. *Progress* (Progresso): um termo bem tipado não está *stuck*[5] (ou é um valor ou pode ser reduzido).

[3]Haskell tem o valor $\perp$, mas não o tipo $\perp$. No Sistema F, por outro lado, $\perp \equiv \forall a.a$.

[4]Esta função não faz parte do prelúdio do Haskell.

[5]Um estado *stuck* é quando um termo está na forma normal, mas não é um valor (algum termo de um tipo constante).

2. *Preservation* (Preservação): se um termo bem tipado leva um passo de redução, então o termo resultante é também bem tipado.

Apesar da ausência de uma prova, acredita-se que Haskell é uma linguagem *type safe* pois é baseada no sistema Damas-Milner, o qual tem uma prova [1] que resultou no conhecido slogan de Milner: "Programas bem tipados não dão errado" (Tradução do autor). Por outro lado, o uso da função `unsafeCoerce` certamente invalida a sua propriedade *type safe* e também a tornaria trivial se já não fosse. Vale mencionar que *Type safety* é uma condição ortogonal e mais fraca que trivialidade e consistência, apesar de que todas estão relacionadas com situações não ideais em sistemas de tipos. O obejtivo de Haskell é ser uma linguagem de programação de propósito geral e não ser uma lógica, como por exemplo o assistente de provas Coq.

4.2 *Theorems for free* em Haskell

Theorems for free! (ou Teoremas de graça!) é o nome de um artigo de Philip Wadler [2], cuja ideia central é a possibilidade de derivar teoremas a partir de funções polimórficas (paramétricas). Esse resultado é referenciado, também, como *parametricity* e é uma reformulação do Teorema da Abstração de Reynolds, que está presente em [19]. *Parametricity* garante, por exemplo, que se fornecida uma função `f :: [a] -> [a]` é possível deduzir o seguinte teorema: para todos os tipos `a` e `b`, para toda lista `xs :: [a]`, e para toda função total `g :: a -> b`, tem-se a igualdade

```
map g (f xs) = f (map g xs),
```

que pode ser reescrita, omitindo o argumento lista `xs`, como

```
(map g) ∘ f = f ∘ (map g),
```

o que, via Curry-Howard, alega que a composição de provas é comutativa sobre a suposição do tipo da função `f`.

A explicação intuitiva sobre isso é que o tipo de uma função diz algumas coisas sobre o seu comportamento. Do seu tipo, deduz-se que a função `f ::[a] -> [a]` pode reordenar seus elementos, eliminá-los, repeti-los, e aplicar uma função de tipo `a -> a`, mas jamais aplicar uma função soma ou adicionar um elemento (de um tipo específico), pois isso iria, respectivamente, restringir e instanciar o tipo de `f`. Por exemplo, toma-se `f` como a função `reverse :: [a] -> [a]` e `g` como `(+1) :: Num a => a -> a`. Logo,

```
map (+1) (reverse [1, 2, 3]) = reverse
 ↪ (map (+1) [1, 2, 3]) = [4, 3, 2].
```

Como existe `undefined` em Haskell, então é possível tomar `f` como

```
insertUndefined :: [a] -> [a]
insertUndefined xs = undefined : xs,
```

o que quebra o comportamento esperado de uma função de tipo `[a] -> [a]`, pois insere um elemento. No entanto, o teorema de `f` é mantido, pois a computação sempre resulta em $\bot$, não importa `xs` e `g`. Por exemplo,

```
map (+1) (insertUndefined [1, 2, 3]) =
insertUndefined (map (+1) [1, 2, 3]) =
 ↪ ⊥.
```

Inconvenientemente, `undefined` pode ser usado para quebrar o teorema em questão, basta tomar `f` e `g` como as funções constantes abaixo:

```
const_f :: [a] -> [a]
const_f _ = [undefined]

const_g :: a -> Int
const_g _ = 1
```

Assim, tem-se

```
map const_g (const_f [1, 2, 3]) = [1]
const_f (map const_g [1, 2, 3]) = ⊥.
```

Isso acontece, pois a avaliação preguiçosa (*lazy evaluation*) do Haskell esconde o erro no primeiro caso. A avaliação estrita (*eager evaluation*) poderia recuperar o teorema no entanto.

Ainda, Haskell tem outra função que enfraquece *Theorems for free* quando usada. Essa é a função `seq :: a -> b -> b` e é uma das funções primitivas para a introdução da avaliação estrita. De certa maneira, `seq` magicamente avalia de maneira estrita o seu primeiro argumento e retorna o segundo. "Magicamente" porque não é possível definir `seq` próprio Haskell, já que sua implementação é feita pela introdução de uma dependência virtual de valores entre o resultado `y` e o argumento `x`, como em:

```
equal :: Eq a => a -> b -> b
equal x y = if x == undefined then y
 ↪ else y,
```

que avalia o argumento `x` antes de retornar `y`, com a diferença que `seq` não precisa ter o primeiro argumento como instância da classe **Eq**. Há como definir `seq` conforme abaixo:

```
seq ⊥ b = ⊥
seq _ b = b
```

Porém, `seq` não é capaz de resolver o problema da parada para determinar se o primeiro argumento é $\bot$. Então, tenta-se a redução do primeiro argumento. Se o primeiro argumento tem *weak head normal form* (WHNF) ou forma normal, então `seq` retorna `b`, caso contrário fica em loop ou retorna uma exceção.

Com `seq` é possível definir a função

```
tail_seq :: [a] -> [a]
tail_seq (x:xs) = seq x xs,
```

então tomando `tail_seq` como `f` e `const 1` como `g`, o teorema anterior é quebrado, pois

```
map (const 1) (tail_seq [1 `div` 0]) =
 ↪ ⊥
tail_seq (map (const 1) [1 `div` 0]) =
 ↪ [].
```

5. Análise Categórica do Sistema de Tipos de Haskell

Um Cálculo Lambda Tipado projeta uma categoria cartesiana fechada [4], portanto respeita regras básicas, como: a existência da identidade para qualquer objeto e a associatividade da composição de morfismos. Além disso, ser cartesiana fechada implica na existência de todos os produtos finitos e de todos os objetos exponenciais. Consequentemente, esse modelo computacional é bem comportado e tem características interessantes como funções de ordem superior. Por esses mesmos motivos, uma linguagem de programação real que projete uma categoria cartesiana fechada é uma boa meta. Nesta seção é investigado se Haskell seria essa linguagem. Nota-se que Haskell frequentemente utiliza termos trazidos da Teoria das Categorias, portanto pode-se supor que exista uma categoria que permeia a linguagem. A página *Wiki* do Haskell [20] afirma quanto à existência de uma categoria chamada **Hask**:

> Os objetos de **Hask** são tipos de Haskell e os morfismos dos objetos **A** a **B** são funções de Haskell do tipo `A -> B`. O morfismo identidade do objeto **A** é `id :: A -> A`, e a composição dos morfismos `f` e `g` é `f . g = x -> f (g x)`. (Tradução do autor)

Em seguida, a *Wiki* aponta o problema com as seguintes duas funções:

```
undef1 :: a -> b
undef1 = undefined

undef2 :: a -> b
undef2 = \_ -> undefined
```

Observa-se que `undef2` está em WHNF e `undef1` não tem forma normal. Apesar de que `undef1 1 = undef2 1`, ambas não têm o mesmo valor semântico, pois

```
seq undef1 1 = ⊥
seq undef2 1 = 1
```

e, concomitantemente, `undef1 . id = undef2`, o que implica em `undef1 . id` $\neq$ `undef1`. Assim, diz-se que `undef1 . id` e `undef1` não são observacionalmente equivalentes, pois no contexto do primeiro argumento de `seq` não tem o mesmo valor. Como uma maneira de contornar esse problema, a *Wiki* afirma que duas funções `f` e `g` são a mesma se para todo `x` tem-se `f x` $=$ `g x`. Por fim, conclui-se que `undef1` e `undef2` são valores diferentes, mas representam a mesma função/morfismo em **Hask**. Isso significa que dois programas representam o mesmo morfismo, porém são dois objetos exponênciais diferentes.

Como apontado por Andrej Bauer em seu blog [21], em Haskell não há semântica operacional que determina a equivalência observacional de dois termos e a relação de igualdade precisaria ser definida para todos os construtores de tipo, inclusive para tipos recursivos. Até que esse trabalho seja apresentado, não é possível dizer que há categoria **Hask**. Infelizmente, isso mostra falta de rigor matemático na Wiki do Haskell.

Por ora **Hask** é apenas uma possibilidade, mas talvez seja certo falar de uma categoria num subconjunto da linguagem Haskell. Uma opção seria eliminar a função `seq` e todas as outras funções definidas através dela, consequentemente não haveria funções estritas. Outra opção seria trocar a avaliação preguiçosa pela estrita, porém isso causaria mudanças na flexibilidade da linguagem, como a impossibilidade de criar listas infinitas. Considerar apenas funções totais (inclusive removendo o operador de ponto fixo) eliminaria o valor $\perp$, mas é uma mudança que restringe severamente a finalidade da linguagem. A escolha feita pela *Wiki* do Haskell é considerar funções totais, o que deve permitir definir uma categoria cartesiana fechada. Assim, esta é a opção escolhida para ser analisada.

A categoria *Platonic* **Hask**, denotada por **PHask**, tem uma definição similar a **Hask**, porém permite apenas funções totais, assim exclui-se o operador de ponto fixo `fix :: (a -> a) -> a` e, portanto, funções com recursão geral não são consideradas.

Definição 5.1. Categoria **PHask**

1. Obj_{PHask} é o conjunto de todos os tipos de Haskell. Vide que isso inclui todos os tipos algébricos, inclusive o sem valor **`data Empty`** e o com um único valor **`data Unit = Unit`** (ou **`data () = ()`**).
2. Mor_{PHask} é o conjunto de todas as funções em Haskell (tipáveis) que são totais. Além disso, um morfismo `f :` **`A`** $\rightarrow$ **`B`** representa a classe de equivalência de funções do tipo **A** ao tipo **B** que tem o mesmo mapeamento.
3. ∂_0 e ∂_1 são as funções que levam, respectivamente, uma função ao seu tipo de origem e destino.
4. A composição é dada pela função `(.) :: (b -> c) -> (a -> b) -> a -> c` e a prova da associatividade é a mesma utilizada em **Set**.
5. O morfismo identidade é gerado pelas funções identidade `id :: a -> a`, que pelo polimorfismo garante que todo objeto (tipo) tem um morfismo (função) identidade. Observa-se que **PHask**, diferente de **Hask**, não acontece `undef1 . id` $\neq$ `undef1`. Uma vez que não há valor $\perp$, não há diferença semântica, em especial a contextual em `seq`, entre `f` e `f . id` para todo `f`.

Teorema 5.1. A categoria **PHask** é uma categoria cartesiana fechada.

O objeto terminal é o tipo `()` ou qualquer outro tipo com um único valor, como por exemplo o **`Unit`**. Devido à totalidade das funções, de qualquer outro objeto (tipo) **A** existe

um único morfismo para `()`. O produto pode ser definido de diversas maneiras através de dados algébricos, aos quais são equivalentes. Utiliza-se o dado tupla padrão: o produto é dado pelos morfismos das funções `fst :: (a, b) -> a` e `snd :: (a, b) -> b`, e pelo tipo `data (a, b) = (,) {fst :: a, snd :: b}`. Assim, para qualquer tipo `r` e para quaisquer funções `f :: r -> a` e `g :: r -> b` deve existir uma única função `u :: r -> (a,b)` capaz de fazer o diagrama abaixo comutar.

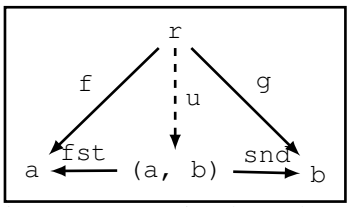

Figura 5. Diagrama comutativo do produto em Haskell.

A melhor função `u` que garante a comutatividade desse diagrama somente pode ser:

```
u :: r -> (a, b)
u r = (f r, g r).
```

O objeto exponencial em Haskell é formado pelo objeto `b -> c`[6] e pelo morfismo `eval`, onde `b` e `c` são tipos quaisquer e `eval` é gerado pela função

```
eval :: (b -> c, c) -> c
eval (f, x) = f x,
```

pois para todo tipo `a` e função `g :: (a, b) -> c`, existe um único morfismo `curry g`, onde

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f = \x -> \y -> f (x, y),
```

que faz o diagrama da Figura 6 comutar.

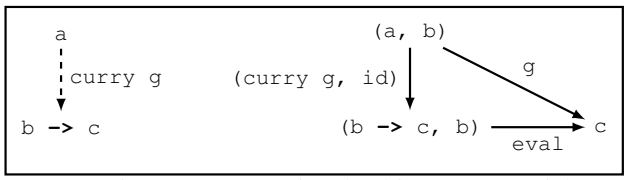

Figura 6. Diagrama comutativo do objeto exponencial em Haskell.

O objeto exponencial, em Haskell, é sobre a possibilidade de tratar funções como objetos, ou seja, valores de ordem superior e, também, sobre a equivalência `(a, b) -> c` $\equiv$ `a -> b -> c`. No entanto, em Haskell, os argumentos de uma função n-ária geralmente não são representados por uma n-tupla, pois *currying* está implícito nos vários argumentos de uma função. Assim, uma função de aridade maior que um pode ser reescrita como uma função unária que retorna uma função. Portanto, a função `const :: a -> b -> a` gera o morfismo m_1 `: a` $\to$ `b -> a`. A função `uncurry :: (a -> b -> c) -> (a, b) -> c` é a inversa de `curry`, a qual permite construir `uncurry const :: (a, b) -> a` e gerar o morfismo m_2 `: (a, b)` $\to$ `a`, logo `(a, b)` $\to$ `a` $\equiv$ `a` $\to$ `b -> a`.

[6] A notação de objeto exponencial `a -> b` é equivalente à b^a.

5.1 Análise Categórica de Classes de Tipos

As classes de tipos do Haskell são utilizadas para definir algumas estruturas matemáticas, como relações (por exemplo: equivalência e ordem), semigrupos, monoides, funtores e mônadas. Esta subseção analisa as classes de tipos de monoide e funtor, e verifica se são as estruturas que alegam ser.

Um monoide é uma estrutura algébrica com uma única operação binária, associativa e com um elemento neutro. Ou visto como uma categoria: um único objeto, um conjunto de morfismos, onde o morfismo identidade é o elemento neutro e composição é a operação binária associativa. Em suma, os monóides obedecem as seguintes leis:

$$(x * y) * z = x * (y * z)$$
$$e * x = x$$
$$x * e = x,$$

onde $*$ é a operação binária e e é o elemento neutro.

Na classe **Monoid** a função `mempty` é o elemento neutro e a função `mappend` é a função binária. `mconcat` é uma função opcional e que já tem uma implementação padrão que aplica a operação binária numa lista de elementos.

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Um exemplo de monóide é o tipo lista `[]` e a sua operação de concatenação `(++)`, o qual é dado como instância de **Monoid** conforme abaixo.

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Pela definição de uma classe há apenas como garantir a existência de funções (morfismos ou objetos) de uma estrutura. As leis dos monóides não são garantidas na assinatura da classe. Tão pouco a classe **Eq** garante que uma instância respeita as leis da relação de equivalência (reflexividade, transitividade e simetria). Assim, é possível definir

```
instance Num a => Monoid a where
  mempty = 0
  mappend = (-),
```

onde `(-)` não é uma operação associativa e `0` não funciona sempre como o elemento neutro.

A classe **Functor** indica, por seu nome, representar as estruturas categóricas dos funtores, portanto deve ser um mapeamento entre categorias, o qual preserva origem e destino dos morfismos, identidade dos objetos e a composição. Mais precisamente um funtor é uma tupla de funções: uma que leva objetos em objetos e outra que leva morfismos em morfismos. O mapeamento em questão é de **PHask** (uma vez que existência de **Hask** não foi provada) para uma subcategoria de **PHask**, denotada por **Func**, cujos objetos são os tipos que são instâncias de **Functor** e os morfismos são funções definidas entre estes tipos.

Um construtor de tipos é uma função que recebe tipos como argumentos e retorna algum tipo, por exemplo o construtor **Maybe** recebe um tipo `a` e retorna **Maybe** `a`. Uma função cujo mapeamento é entre tipos não tem um tipo, na realidade tem algo chamado de *kind*. O *kind* de **Maybe** é `* -> *`, onde `*` pode ser entendido como o tipo do tipo. Assim, construtores de tipos são o primeiro elemento da tupla do mapeamento funtorial.

O segundo elemento é a função `fmap :: (a -> b) -> f a -> f b` que é declarada na classe:

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b.
```

Por exemplo, o tipo lista um objeto na classe **Func** e a sua instância como **Functor** garante a existência de morfismos entre listas.

```
instance Functor [] where
  fmap = map
```

A assinatura de `fmap` garante a preservação da origem e do destino, porém a identidade e a composição não são asseguradas. Seria necessário que

```
fmap id = id
fmap (f . g) = fmap f . fmap g.
```

Mais uma vez, classes de tipos não são capazes de garantir tais propriedades na instância da classe. Provavelmente as demais classes de tipo, como mônadas e comônadas[7], também não são realmente as estruturas matemáticas que são indicadas por seus nomes. Isso é uma escolha de design da linguagem, que tem a preocupação de ser uma linguagem de programação de uso geral e não um sistema de lógica.

5.2 Transformações Naturais em I

Uma Transformação Natural T é um homomorfismo entre funtores, ou seja, converte um funtor F_1 em outro F_2 ao mapear os objetos e morfismos da imagem de F_1 nos objetos e morfismos da imagem F_2. Assim, dado duas categorias A, B e dois funtores F_1, F_2 de A para B (ver Figura 7), tem-se que todos os objetos a na categoria A têm suas imagens em B representadas, respectivamente, por $F_1\,a$ e $F_2\,a$. Os morfismos da transformação natural T que mapeiam, para todo a em A, $F_1\,a$ em $F_2\,a$ são chamados de m_a (também conhecidos como componentes de T). Os morfismos f da categoria A mapeados por F_1 e F_2 são chamados de $F_1\,f$ e $F_2\,f$ respectivamente. Então, uma Transformação Natural T são os morfismos $m_a : F_1\,a \to F_2\,a$ e $m_b : F_1\,b \to F_2\,b$ associados à todos os objetos a, b da categoria A e os morfismos $F_1\,f : F_1\,a \to F_1\,b$ e $F_2\,f : F_2\,a \to F_2\,b$ associados todos os morfismos $f : a \to b$ na categoria A, tais que satisfazem a condição de naturalidade

$$m_b \circ F_1\,f = F_2\,f \circ m_a,$$

ou que fazem o diagrama D da abaixo comutar.

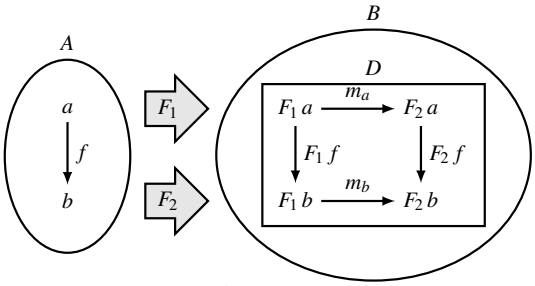

Figura 7. Representação de uma Transformação Natural entre categorias A e B.

Transformações Naturais são endofuntores (funtores de uma categoria para ela mesma) e, como já apresentado, funtores são representados por construtores de tipos e pela função `fmap`. O objeto $F_1 a$ é um tipo de dado parametrizado por a com uma implementação de `fmap`, como por exemplo `[a]` e **Maybe** `a`. Portanto, uma Transformação Natural deve mapear ter uma função $m_a : F_1\,a \to F_2\,a$, ou seja, uma função de tipo `ma :: F1 a -> F2 a` e que respeita a condição de naturalidade. Por exemplo, tomando m_a como a função

```
safeHead :: [a] -> Maybe a
safeHead []     = Nothing
safeHead (x:xs) = Just x
```

de fato respeita a naturalidade

```
safeHead . fmap f = fmap f . safeHead
```

A prova disso segue por análise de caso. Caso a lista seja vazia tem-se

```
safeHead . fmap f = fmap f . safeHead
↪  = []
```

ou caso a lista seja não-vazia, então

```
safeHead . fmap f = fmap f . safeHead
↪  = Just (f x)
```

Na realidade, não é necessário provar a condição de naturalidade para todos os tipos de Haskell pois isso é um *Theorem for Free*. Porém, como já mostrado, isso pode ser quebrado com o `undefined`.

O uso de Transformações Naturais acontece em Haskell quando deseja-se converter dados entre tipos que são instâncias da classe **Functor**.

[7] O dual de mônada.

5.3 Mônadas são Apenas Monoides em FHask

A abordagem desta subseção será oposta as anteriores, primeiro será apresentado o que são mônadas em Haskell e posteriormente em Teoria das Categorias. Haskell é uma linguagem de programação pura, funções são funções matemáticas, o que permite pensamento equacional sobre programas, torna a ordem de avaliação irrelevante e, assim, pode se beneficiar da avaliação preguiçosa. A ordem de avaliação em linguagens impuras pode alterar o valor final da computação. Considere a função impura `getChar :: Char`, a qual lê um carácter da linha de comando e o retorna como um **`Char`**, e o seguinte programa

```
get2Chars = [first, second] where
                    first  = getChar
                    second = getChar
```

Esse programa lê duas vezes a entrada do usuário e coloca os valores numa lista, porém dependendo da ordem da avaliação `first` pode não ser o primeiro carácter digitado. Não há garantia que o primeiro elemento da lista vai ser avaliado antes do segundo. Uma maneira simples de forçar a ordem de avaliação seria utilizar um argumento em `getChar` que represente a sequência de entrada e com isso criar uma dependência de valores na avaliação.

```
getChar :: Int -> (Int, Char)

get2Chars i0 =
             ([first, second], i2) where
               (first, i1)  = getChar i0
               (second, i2) = getChar i1
```

Retornar `i2` é necessário para evitar o mesmo problema anterior, assim há como determinar a ordem de avaliação de `get2Chars` na definição de outra função.

Esse exemplo mostra como uma função impura pode ser representada de maneira pura com o uso de uma informação extra sobre a ordem da avaliação. Por outro lado, o seu uso é menos prático, pois é necessário extrair uma informação da dupla com *pattern matching* e passar o valor adiante. Seria conveniente que a extração e passagem de argumentos acontecessem de maneira implícita.

Ao invés de utilizar uma dupla para representar a leitura de valores, considera-se o tipo [8]

```
type IO a = Int -> (a, Int),
```

o qual representa de maneira polimórfica as interações de entrada e saída. Logo, `getChar :: IO Char`.

A computação implícita é feita pelo operador `(>>=)` (*bind*) e `return`, cujas implementações para o tipo **`IO`** `a` são

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(action1 >>= action2) world0 =
       (b, world2) where
         (a, world1) = action1 world0
         (b, world2) = action2 a world1

 return :: a -> IO a
 return x = \s -> (x, s),
```

onde `(>>=)` funciona como um operador de composição para os dados do tipo **`IO`** `a` e `return` coloca um valor dentro do tipo **`IO`** `a`. Consequentemente, é possível reescrever o programa anterior de uma maneira puramente funcional e com passagem implícita de argumento:

```
get2Chars = getChar >>= \x -> getChar
 ↪  >>= \y -> return [x, y].
```

A notação **`do`** é um açúcar sintático para uma representação similar à um código imperativo:

```
get2Chars = do
            x <- getChar
            y <- getChar
            return [x, y]
```

Haskell introduz mônadas como uma classe de tipos com três operações:

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

onde o operador `(>>)` pode ser definido como

```
(m1 >> m2) = m1 >>= \_ -> m2.
```

Além de **`IO`** `a` diversos tipos são instâncias de **`Monad`**, como `[a]`, **`Maybe`** `a`, **`Either`** `a b`, **`State`** `s` e **`Rand`** `g`. Em resumo, mônadas em Haskell servem para compor computações não triviais, as quais possuem algum tipo de informação adicional ou efeito colateral.

A ideia de mônada em Teoria das Categorias é similar a de monoide na Teoria dos Conjuntos, é uma generalização. Então, para ambos é necessário uma operação binária (composição de endfuntores) associativa e um elemento (endofuntor) identidade (para esquerda e direita). Mac Lane[22] resume essa ideia na famosa frase:

> Tudo dito, uma mônada em X é apenas um monoide na categoria dos endofuntores de X, com o produto $\times$ substituído pela composição de endofuntores e o conjunto unitário pela endofuntor identidade.

Formalmente, uma mônada para uma dada categoria X é uma tripla composta por um endofuntor $T : X \to X$ e duas transformações naturais $\eta : I \to T$ e $\mu : T \star T \to T$. Aqui I é o endofuntor identidade e $\star$ é uma operação de composição de endofuntores. Essa tripla deve respeitar as condições de coerência:

[8] Para que o código fique mais legível, nesse exemplo o tipo IO é declarado como um sinônimo. Na implementação real, para que a sobrecarga seja possível é necessária a declaração como um tipo algébrico com o respectivo construtor de dado. Além disso, utiliza-se um tipo **`RealWolrd`** no lugar de **`Int`**.

- $\mu \circ T\mu = \mu \circ \mu T$, onde $\circ$ é a composição de transformações naturais;
- $\mu \circ T\eta = \mu \circ \eta T = 1$, onde 1 é a transformação natural identidade.

A primeira condição em prosa: o endofuntor T aplicado na transformação μ ($T\mu$ atua na imagem de T) composto com μ é o mesmo que aplicar T e compor resultado com $\mu \circ mu$. A segunda condição: o endofuntor T aplicado na transformação η e composto com μ é a transformação natural identidade, assim como aplicar T e compor resultado com $\eta \circ \mu$. As condições podem ser representadas pelos, respectivos, diagramas comutativos da Figura 8, parametrizados por uma categoria X.

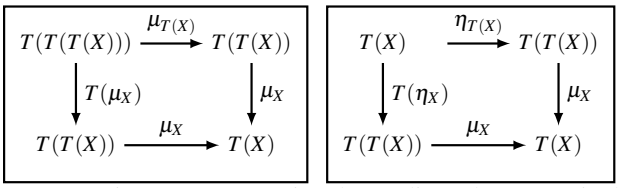

Figura 8. Diagrama comutativo da condição de coerência de mônada.

O endofuntor T em Haskell é apenas um construtor de tipos `M a` que é uma instância da classe **`Functor`**. A primeira transformação natural leva o endofuntor identidade num outro endofuntor T, mas como em Haskell um construtor de tipos que leva um tipo nele mesmo é o próprio tipo, então tem-se que η é `return :: a -> M a`. A segunda transformação natural leva um endofuntor T composto consigo mesmo à um endofuntor T, que traduz-se para Haskell como `join :: M (M a) -> M a`.

A definição categórica tem `join` como evidente diferença em relação a classe **`Monad`** apresentada anteriormente. Há uma representação alternativa da classe de mônadas:

```
class Monad (m :: * -> *) where
  join :: m (m a) -> m a
  return :: a -> m a,
```

a qual é equivalente a anterior pois,

```
join :: Monad m => m (m a) -> m a
join x = x >>= id

(>>=) :: Monad m => m a -> (a -> m b)
 ↪ -> m b
x >>= f = join (fmap f x).
```

A categoria **PHask** tem vários objetos que são endofuntores: todos os construtores de tipos que são instância da classe **`Functor`**. Todos os funtores em Haskell são endofuntores, pois sempre mapeiam da categoria **PHask** para **PHask**, por isso têm o *kind* `* -> *`. Considere uma subcategoria chamada **FHask**, cujos objetos são os endofuntores de **PHask** e os morfismos são transformações naturais. Tudo dito, uma mônada é apenas um monoide na categoria **FHask** (ver figura 9).

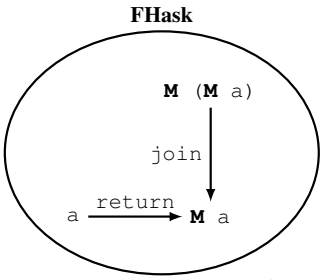

Figura 9. Monoide na categoria **FHask**.

Como realizado anteriormente, cabe verificar se a classe **`Monad`** garante que suas instâncias são realmente mônadas. As condições de coerência devem ser satisfeitas em termos de Haskell, ou seja,

```
join . fmap join = join . join
join . fmap return = join . return =
 ↪ id,
```

as quais, como anteriormente, não podem ser garantidas pelas classes de tipos e, portanto, ao implementar-se uma nova mônada é decisão do programador respeitá-las.

6. Considerações Finais

Definiu-se uma pequena formalização do sistema de tipos de Haskell, a qual reflete as suas principais funcionalidades. Argumentou-se que esse sistema, por Curry-Howard, corresponde à um fragmento da Lógica Proposicional de Segunda Ordem Intuicionista ainda menor que o Sistema-F. Um dos principais aspectos lógicos desse sistema é não representar uma lógica consistente, pois qualquer tipo é habitado e, assim, todas as proposições são teoremas. No entanto, Haskell objetiva ser uma linguagem de programação de uso geral e não um sistema de lógica. Logo não há a necessidade de ser uma lógica consistente e não trivial, somente é necessário que seja *type safe*.

A extração de teoremas a partir de tipos, conhecido por *Theorems for free*, é uma característica interessante para Haskell, pois mostra como o tipo delimita quais funções o habitam e quais são os possíveis comportamentos destas funções. Porém, tal característica é enfraquecida devido a termos como `undefined` e `seq`. Esse enfraquecimento não inviabiliza totalmente utilizar *Theorems for free*, mas apenas aponta que é necessário utilizá-lo com ressalvas.

No que diz respeito à análise categorial, mostrou-se que a nomenclatura de Teoria das Categorias é utilizada apenas como inspiração e não reflete necessariamente os verdadeiros conceitos categoriais. A ausência de uma semântica operacional em Haskell dificulta a argumentação em favor da existência de uma categoria e como alternativa delimita-se uma sub-linguagem total. Transformações Naturais representam funções entre tipos que são funtores e que a condição

de naturalidade pode ser obtida como um *Theorem for free*. Utilizou-se de um exemplo prático para deduzir uma mônada de **`IO`** e para mostrar que mônadas servem para compor computações não triviais. Relacionou-se a definição formal de mônada com a classe **`Monad`** e como mônadas são apenas monoides na categoria **FHask**.

Author contributions

- Rafael Castro G. Silva: Research, definition of methodology and writing.
- Karina Girardi Roggia: Scope definition and revision on category theory.
- Cristiano Damiani Vasconcellos: Scope definition and review on type systems and Haskell.

References

[1] DAMAS, L.; MILNER, R. Principal type-schemes for functional programs. In: DEMILLO, R. (Ed.). *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, USA: ACM, 1982. (POPL '82).

[2] WADLER, P. Propositions as types. *Commun. ACM*, v. 58, n. 12, p. 75–84, 2015.

[3] HOWARD, W. A. The formulas-as-types notion of construction. In: SELDIN, J. P.; HINDLEY, J. R. (Ed.). *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. 1. ed. New York, USA: Academic Press, 1980, (POPL, '80). p. 479–490.

[4] LAMBEK, J.; SCOTT, P. *Introduction to Higher-Order Categorical Logic*. 1. ed. Cambridge, UK: Cambridge University Press, 1988. v. 1. (Cambridge Studies in Advanced Mathematics, v. 1).

[5] JONES, S. P. *Haskell 98 language and libraries : the revised report*. 1. ed. Cambridge, UK: Cambridge University Press, 2003.

[6] WADLER, P.; BLOTT, S. How to make ad-hoc polymorphism less ad hoc. In: YORK, C. N. (Ed.). *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1989. (POPL, '89).

[7] KFOURY, A. J.; TIURYN, J.; URZYCZYN, P. The undecidability of the semi-unification problem. In: ORTIZ, H. (Ed.). *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*. Baltimore, USA: ACM, 1990. (STOC, '90).

[8] FERNANDES, F. L. *O Isomorfismo de Curry-Howard via Teoria de Categorias*. 2011. Monography.

[9] MOGGI, E. Computational lambda-calculus and monads. In: IEEE. *Logic in Computer Science, 1989*. Pacific Grove, USA: IEEE, 1989. (4).

[10] WADLER, P. Comprehending monads. In: KAHN, G. (Ed.). *Proceedings of the 1990 ACM conference on LISP and functional programming*. New York, USA: ACM, 1990. v. 1.

[11] WADLER, P. Monads for functional programming. In: JEURING, E. M. J. (Ed.). *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK: Springer-Verlag, 1995. v. 1.

[12] GENTZEN, G. Investigations into logical deduction. In: SZABO, M. E. (Ed.). *The Collected Papers of Gerhard Gentzen*. 1. ed. Amsterdam: North-Holland, 1969. v. 1, p. 68–213.

[13] MENEZES, P.; HAEUSLER, E. H. *Teoria das Categorias para Ciência da Computação*. 1. ed. Porto Alegre, Brazil: Editora Sagra Luzzatto, 2001. v. 12. (Livros, v. 12).

[14] SØRENSEN, M. H. B.; URZYCZYN, P. *Lectures on the Curry-Howard Isomorphism*. 1998.

[15] GIRARD, J.-Y. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Tese (Doutorado) — PhD thesis, Université Paris VII, Paris, France, 1972.

[16] REYNOLDS, J. C. Towards a theory of type structure. In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. Berlin, Heidelberg: Springer-Verlag, 1974. (LNCS, v. 29).

[17] COSTA, N. C. d.; ABE, J. M. Paraconsistência em informática e inteligência artificial. *Estud. Av.*

[18] PIERCE, B. C. *Types and Programming Languages*. 1st. ed. Cambridge, USA: The MIT Press, 2002. v. 1.

[19] REYNOLDS, J. C. Types, abstraction and parametric polymorphism. In: *IFIP Congress*. Berlin, Germany: Springer, 1983. (LNCS, v. 598).

[20] HASKELLWIKI. *HaskellWiki*. 2017. Https://wiki.haskell.org/Hask.

[21] BAUER, A. *Mathematics and Computation - Hask is not a category*. 2017. Http://math.andrej.com/2016/08/06/hask-is-not-a-category.

[22] LANE, S. M.; LANE, S. M. *Categories for the Working Mathematician*. 2. ed. Berlin, Germany: Springer New York, 1998. v. 5. (Graduate Texts in Mathematics, v. 5).