Rafael Castro Gonçalves Silva

Visão Categórica do Sistema de Tipos de Haskell

Joinville

2017

UNIVERSIDADE DO ESTADO DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Rafael Castro Gonçalves Silva

VISÃO CATEGÓRICA DO SISTEMA DE TIPOS DE HASKELL

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Karina Girardi Roggia
Orientadora

Joinville, Junho de 2017

VISÃO CATEGÓRICA DO SISTEMA DE TIPOS DE HASKELL

Rafael Castro Gonçalves Silva

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação Integral do CCT/UDESC.

Banca Examinadora

Karina Girardi Roggia - Doutora (orientadora)

Rodrigo Machado - Doutor

Cristiano Damiani Vasconcellos - Doutor

Cláudio César de Sá - Doutor

Agradecimentos

Agradeço à minha família que me deu suporte durante toda a minha graduação, especialmente minha mãe que enfrentou várias barreiras para que não me faltasse nada. Agradeço aos meus amigos pelas risadas que amenizaram a monotonia do dia a dia. Por fim, agradeço à minha orientadora Profa. Karina Roggia, cuja paixão pela Ciência da Computação me inspirou em todos os momentos deste trabalho.

There's nothing more practical than a good theory.

Kurt Lewin

Resumo

Sistemas de tipos de linguagens de programação estão cada vez mais sofisticados e, em alguns casos, são baseados em formalismos como Teoria dos Tipos, Cálculo Lambda e Teoria das Categorias. Estas teorias, que foram criadas com propósitos distintos no século passado, servem como fundamentos para pesquisas em sistemas de tipos. O objetivo deste trabalho é explorar as teorias mencionadas e relacioná-las com o sistema de tipos, assim este trabalho contém diversas definições dessas teorias, argumentações sobre as vantagens que elas trazem e uma análise do sistema de tipos de Haskell por meio de uma visão categórica.

Palavras-chave: *Sistemas de tipos, teoria das categorias, cálculo lambda, Haskell, correspondência Curry-Howard-Lambek*

Abstract

Type systems in programming languages are becoming more and more sophisticated and in some cases are based on formalisms such as Type Theory, Lambda Calculus and Category Theory. These theories, which were created for different purposes in the the last century, serve as foundations for research into type systems. The purpose of this thesis is to explore theories mentioned and how to relate them with type systems, so this work contains several definitions of these theories, arguments about the advantages they bring, and an analysis of Haskell's type systems by means of a categorical view.

Keywords: *Type systems, Category Theory, lambda calculus, Haskell, curry-howard-lambek correspondence*

Lista de Figuras

2.1 Teorema de Church-Rosser para a Redução β. 17

2.2 Prova do Teorema de Church-Rosser para Equivalência β. Na esquerda o caso (b) e na direita o caso (c) 19

3.1 Algoritmo W: representação sob a forma de função. 42

3.2 Operações aritméticas sobrecarregadas com classes de tipo. 45

3.3 Classe de tipo das operações aritméticas traduzida para tipo de dado. . . . 46

3.4 Exemplo de subclasse para operações aritméticas. 46

3.5 Sintaxe de termos e tipos em *Ad-Hoc-Damas-Milner* (AHDM). 48

4.1 Mapeamentos possíveis para as funções de tipo $1 \to bool$. 58

4.2 Representação parcial da categoria de uma linguagem de programação. . . 59

4.3 Diagrama Comutativo. 61

4.4 Diagrama comutativo do produto entre A e B. 65

4.5 Diagrama Comutativo do coproduto entre A e B. 66

4.6 Representação de um *Cone* sobre um diagrama D. 67

4.7 Diagrama sem objetos e morfismos (esquerda), diagrama com dois objetos e dois morfismos paralelos (meio), e diagrama com dois objetos e sem morfismos (direita). 67

4.8 Maior divisor comum entre 36 e 90. 68

4.9 Produto como limite do diagrama com dois objetos. 69

4.10 Conjunto A_0 equalizador das funções f e g. 70

4.11 Esquema com função binária g e espaço de função B^A. 72

4.12 Esquema com função $eval_g$. 73

4.13 Diagrama do objeto exponencial. 73

4.14 Diagrama comutativo do objeto exponencial. 74

5.1 Diagrama comutativo do produto entre ϕ e φ. 95

5.2 Diagrama comutativo do objeto exponencial na categoria **CND**. 96

5.3 Diagrama comutativo do produto entre τ e α. 98

5.4 Diagrama comutativo do objeto exponencial na categoria **CTA**. 99

6.1 Diagrama comutativo do produto em Haskell. 109

6.2 Diagrama comutativo do objeto exponencial em Haskell. 110

Lista de Abreviaturas

ADT *Algebraic Data Types*

AHDM *Ad-Hoc-Damas-Milner*

AST *Abstract Syntax Tree*

BHK *Brouwer-Heyting-Kolmogorov*

CCC *Cartesian Closed Category*

CHL *Curry-Howard-Lambek*

DM Damas-Milner

SF *System-F*

TA *Type-Assignment*

HM Hindley-Milner

Sumário

Lista de Figuras **5**

Lista de Abreviaturas **7**

1 Introdução **10**

2 Cálculo Lambda **12**

2.1 Cálculo Lambda Atipado 12

2.2 Cálculo Lambda Simplesmente Tipado 20

2.2.1 Estilo de Church 22

2.2.2 Estilo de Curry 24

2.3 Tipo Principal 31

3 Sistemas de Tipos **34**

3.1 Sistema Damas-Milner 35

3.1.1 Sistema de Tipos de Haskell 42

3.2 Sistema F 51

4 Teoria das Categorias **55**

4.1 Categorias 56

4.2 Diagramas 61

4.3 Subcategorias, Funtores e Categorias de Categorias 62

4.4 Categoria Dual e Dualidade 63

4.5 Objeto Inicial e Objeto Terminal 64

4.6 Produto e Coproduto 65

4.7 Cones e Cocones 66

4.8 Limites e Colimites 68

4.9 Equalizador e Coequalizador 70

4.10 Objeto Exponencial 71

4.11 Categorias Cartesianas Fechadas 75

5 Correspondência Curry-Howard-Lambek 77

5.1 Fragmento da Dedução Natural Intuicionista 77

5.2 Isomorfismo Curry-Howard 81

5.3 Curry-Howard-Lambek 92

5.3.1 Dedução Natural Intuicionista é uma Categoria Cartesiana Fechada 92

5.3.2 Cálculo Lambda Tipado é uma Categoria Cartesiana Fechada . . . 96

6 Análise do Sistema de Tipos de Haskell 100

6.1 Visão Lógica do Sistema de Tipos de Haskell 100

6.1.1 *Theorems for free* em Haskell 103

6.2 Visão Categórica do Sistema de Tipos de Haskell 106

6.2.1 Visão Categórica de Classes de Tipos 110

7 Conclusões 113

Referências Bibliográficas 115

Apêndice 119

A Algoritmo de inferência do sistema TA 119

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 é uma linguagem sucessora de ML e inspirou-se em seu sistema de tipos com inferência de polimorfismo paramétrico. Diferente de ML, Haskell é uma linguagem funcional pura, isso é, livre de efeitos colaterais, que são alterações de estado de memória capazes de influenciar o resultado de computações futuras.

Linguagens funcionais puras têm diversas vantagens como: prescindir exclusão mútua em programação concorrente, poder armazenar computações já realizadas para usos futuros e serem mais fáceis de extrair propriedades de seus programas. Os avanços da computação teórica, em particular da Teoria dos Tipos e Teoria das Categorias, resultaram em novas ideias para sistemas de tipos. Por exemplo, mônada é um conceito categórico possibilitou Haskell tratar entradas e saídas de informações sem causar efeitos colaterais (WADLER, 1995).

O sistema de tipos de ML, Miranda e Haskell são baseados no chamado sistema Damas-Milner, ou Hindley-Milner, cuja característica principal é existência de um algoritmo que sempre encontra o tipo mais geral para um programa, ou o tipo que melhor representa o programa, também conhecido como tipo principal. Além disso, estabelece uma prova construtiva de que a atribuição de tipo é correta. *Propositions as Types* é uma relação entre provas na lógica construtivista e provas em sistemas de tipos (WADLER, 2015), que é principalmente conhecida como Isomorfismo de Curry-Howard.

O Isomorfismo de Curry-Howard estipula que proposições são tipos e provas construtivas (do intuicionismo de Brouwer) são termos do Cálculo Lambda Tipado. A equivalência foi descoberta, em 1969, por William Alvin Howard, com base nos resultados prévios de Haskell Curry e relaciona diretamente o Cálculo Lambda Simplesmente Tipado com a Dedução Natural de Gentzen (HOWARD, 1980). Joachim Lambek descobriu que tal correspondência também está presente dentro das Categorias Cartesianas Fechadas, o que resulta na tripla correspondência Curry-Howard-Lambek (LAMBEK; SCOTT, 1986).

O estudo dessa tripla relação já vem mostrando diversos resultados práticos, por exemplo para descobrir propriedades universais e provas de equivalência entre funções, como feito na generalização de transformações de programas com base numa formalização categórica de provas (TATE; STEPP; LERNER, 2012).

O triplo isomorfismo é um resultado teórico que reúne lógica construtivista, Teoria dos Tipos e Teoria das Categorias. Consequentemente, pode-se questionar como essas três visões estão presentes em um sistema de tipos moderno como o de Haskell. A investigação desse questionamento é o foco deste trabalho.

Os objetivos deste trabalho são: elucidar os principais conceitos de Teoria dos Tipos e Sistemas de Tipos por meio do Cálculo Lambda, definir as principais ideias e resultados de Teoria das Categorias, estabelecer conceitos paralelos entre essas duas teorias, identificar estruturas categóricas e a tripla relação Curry-Howard-Lambek no sistema de tipos de Haskell e, por fim, apresentar exemplos concretos dessas estruturas.

O Capítulo 2 contém definições, provas e discussões a respeito do Cálculo Lambda e suas variações tipadas. O Capítulo 3 apresenta alguns sistemas de tipos. O Capítulo 4 é um conjunto de definições básicas sobre Teoria das Categorias e algumas breves argumentações sobre as vantagens de uma visão categórica de uma estrutura matemática como um sistema de tipos. O Capítulo 5 trata da tripla relação Curry-Howard-Lambek. Por fim, o Capítulo 6 traz a discussão principal: análise categórica do sistema de tipos de Haskell.

2 Cálculo Lambda

Este capítulo dedica-se a entender os fundamentos do Cálculo Lambda Atipado (sem tipos) e sua versão tipada, que são necessários para o futuro desenvolvimento dos sistemas de tipos Damas-Milner e *System-F*.

2.1 Cálculo Lambda Atipado

Cálculo Lambda é um conjunto de sistemas que utilizam como base as concepções originais de Alonzo Church de 1932 (CHURCH, 1932). Inicialmente o sistema foi publicado como um modelo de lógica sem variáveis livres e com quantificadores. Porém, devido à possibilidade de simular o paradoxo de Russel, Church isolou apenas o aspecto funcional do modelo, que viria a se tornar o Cálculo Lambda Atipado (CHURCH, 1933) e, apesar de falhar em seu propósito inicial, mostrou-se ser uma rica teoria de funções.

De maneira geral, o sistema descreve como funções podem ser combinadas para obter outras funções. Funções são conhecidas como abstrações e a sua principal característica é serem de ordem superior (*high-order functions*), isto é, funções podem receber e retornar outras funções. Logo, o estudo das funções e suas composições são o núcleo do Cálculo Lambda Atipado. Por exemplo, descobrir se é possível e como definir uma função que recebe um termo na forma A e retorna outro termo que tem a forma B, ou seja, ratificar a existência de uma regra que permite obter um termo B a partir de um termo A.

Investigar propriedades e teoremas do Cálculo Lambda Atipado fornece informações sobre suas capacidades (ou incapacidades) e possíveis aplicações, que vão além de resultados teóricos como a tese de Church-Turing. Uma dessas são as linguagens de programação funcionais, como LISP, baseadas, indiretamente, no Cálculo Lambda Atipado (CARDONE; HINDLEY, 2006).

A sintaxe define que algumas expressões são válidas e outras não. Nota-se que as funções não precisam de nome e portanto são anônimas. Uma expressão bem formada é classificada como termo λ, que é melhor descrita conforme a indução sobre conjunto

dado abaixo.

Definição 2.1. Termo λ

Dado um conjunto V de variáveis, um termo λ é indutivamente definido por:

(i) Todas as variáveis: $x, y, z, w... \in V$ (átomos) são termos λ

(ii) Se M e N são termos λ, então (MN) é um termo λ. (aplicação)

(iii) Se M é um termo λ e x é uma variável, então $\lambda x.M$ é um termo λ. (abstração λ ou função λ)

Nota 2.1. Por simplicidade, usa-se as seguintes convenções:

1. Letras minúsculas serão utilizadas para representar variáveis e maiúsculas simbolizarão qualquer termo λ.

2. Dois termos λ serão denotados sintaticamente iguais, ou terem a mesma estrutura, por $\equiv$. A definição 2.6 é mais precisa nesta relação.

3. A sintaxe do sistema apenas permite abstrações de uma variável, contudo é possível reescrevê-las como abstrações de várias variáveis e vice-versa. Por exemplo, $h^* = \lambda x.(\lambda y.yx)$ pode ser reescrito como $h = \lambda xy.yx$. Ou seja, h^* recebe um argumento e retorna outra abstração que captura o próximo argumento. A abreviação é denominada *currying*.

4. Os termos λ são associativos para a esquerda, portanto $(\lambda xy.y)ab$ deve entendido como $(((\lambda xy.y)a)b)$. Nesses casos é corriqueira a omissão dos parenteses. Nota-se que em $M(NP)$ omitir os parenteses corresponde a $(MN)P$.

Exemplo 2.1. Termos λ

(a) x

(b) xy

(c) $\lambda x.x$

(d) $\lambda xy.xy$

(e) $x(\lambda x.xy)$

(f) $(\lambda x.x)y$

Abstrações são termos divididos em duas partes por um ponto. A parte anterior ao ponto começa com o símbolo λ e é seguida por uma sequência de argumentos.

O símbolo λ vem da notação $\hat{x}$ para abstração de classe de Whitehead e Russell, o qual Church precisou distinguir, na formulação do Cálculo Lambda, de abstração de função, a qual foi representada por $\wedge x$. Por fim, foi posteriormente modificado para λx por mais fácil impressão. Church alegou que a escolha do símbolo foi mais acidental e que qualquer outro poderia ter sido escolhido (CARDONE; HINDLEY, 2006). A parte posterior ao ponto é conhecida como escopo e é definida abaixo.

Definição 2.2. Escopo da abstração λ

O termo à direita de um ponto na abstração λ é chamado de corpo ou escopo. Na formulação $\lambda x.M$, M é o termo escopo de λx. Assim, $\lambda x.MN$ denota $\lambda x.(MN)$ e não $(\lambda x.M)N$.

Por exemplo, considere $P \equiv \lambda y.w(\lambda h.w)v$. O escopo de λy é $w(\lambda h.w)v$ e o escopo de λh é w.

Variáveis têm o significado matemático tradicional, são valores não especificados e servem para relacionar como os termos capturados pelos argumentos da função serão tratados dentro do escopo da mesma. Condizendo com o que foi supramencionado, o Cálculo Lambda serviu como inspiração para as linguagens funcionais, as quais herdaram essa interpretação tradicional das variáveis. Por outro lado, as linguagens imperativas tratam variáveis como um nome para um lugar na memoria do computador. No Cálculo Lambda aqui tratado, uma variável pode aparecer em três situações.

Definição 2.3. Variáveis livres, ligadas e ligantes

Uma variável x é

(i) ligada se aparece no escopo M de uma abstração $Q \equiv (\lambda x.M)$ de um termo P.

(ii) ligante se aparece nos argumentos de uma abstração.

(iii) livre caso não seja ligada e nem ligante.

O conjunto de todas as variáveis livres de P, chamado de $FV(P)$, é definido indutivamente por:

$$FV(x) = \{x\}$$
$$FV(MN) = FV(M) \cup FV(N)$$
$$FV(\lambda x.M) = FV(M) \setminus \{x\}$$

O papel principal das variáveis livres é segurar posições dentro de um termo, a fim de que durante o processo de substituição essas sejam trocadas por outros termos. Uma substituição é denotada por $[N/x]M$ e deve ser interpretada como a troca de todas as ocorrências livres de x por N no termo M. Múltiplas substituições devem seguir uma ordem de execução conforme indicam os parenteses no termo $([N/x]([M/y](...([H/z]P))))$. As regras abaixo definem o procedimento recursivo de substituição.

Definição 2.4. Regras de Substituição:

(a) $[N/x]x \equiv N$;

(b) $[N/x]a \equiv a$, se $x \not\equiv a$;

(c) $[N/x](PQ) \equiv ([N/x]P[N/x]Q)$;

(d) $[N/x](\lambda x.P) \equiv \lambda x.P$;

(e) $[N/x](\lambda y.P) \equiv \lambda y.P$, se $x \notin FV(P)$;

(f) $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$, se $x \in FV(P)$ e $y \notin FV(N)$;

(g) $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$, se $x \in FV(P)$ e $y \in FV(N)$;

O item (g) serve para evitar que uma variável livre de N seja capturada por λy. Assim, a substituição $[y/x](\lambda y.x)$ resultará em $(\lambda z.y)$ com y livre e não $(\lambda y.y)$.

A substituição é a operação mais fundamental no Cálculo Lambda, pois é usada para definir as reduções. Reduções servem, principalmente, para definir a equivalência entre dois termos e o que a é computação de termo por uma abstração.

Definição 2.5. Redução α ou congruência

Seja um termo P com uma ocorrência de $P \equiv \lambda x.M$ e $y \notin FV(M)$. O ato de substituir P por $Q \equiv \lambda y.[y/x]M$ é chamado de redução α em P. Pode-se aludir que P α-reduz a Q ou que é congruente a Q, ou

$$P \rightarrow_\alpha Q$$

A redução pode ser expressa por:

$$\lambda x.M \rightarrow_\alpha \lambda y.([y/x]M), \text{ se } y \notin FV(M)$$

A redução α estabelece uma relação entre dois termos que possuem a mesma forma ou estrutura, porém utilizam símbolos diferentes nas variáveis. Assim, define-se a relação de equivalência α.

Definição 2.6. Equivalência α

Dois termos são equivalentes se existe uma redução α entre eles. A equivalência α é denotada por $\equiv_\alpha$. Muitos autores usam $\equiv$ no mesmo sentido que $\equiv_\alpha$, pois dois termos α equivalentes podem serem vistos como sintaticamente iguais, ou seja, possuem a mesma construção com base nas definições, mas empregam variáveis diferentes.

Por tratar-se de uma relação de equivalência $\equiv_\alpha$ é reflexiva, transitiva e simétrica. Portanto para quaisquer termos P, Q e R

$$P \equiv_\alpha P \text{ (reflexividade)}$$

$$P \equiv_\alpha Q, Q \equiv_\alpha R \Rightarrow P \equiv_\alpha R \text{ (transitividade)}$$

$$P \equiv_\alpha Q \Rightarrow Q \equiv_\alpha P \text{ (simetria)}$$

Definição 2.7. Redução β

Abstrações λ representam funções e a redução β exprime como aplicá-las a argumentos. Um termo da forma $(\lambda x.M)N$ é chamado de redex β e é interpretado pela substituição $[N/x]M$. A execução da substituição é chamada de contração/redução 1β. A contração 1β de P em P' num passo é denotada por

$$P \rhd_{1\beta} P'$$

Se um termo P pode ser trocado por Q através de um número finito de passos, diz-se que P β-reduz para Q, ou

$$P \rhd_\beta Q$$

Exemplo 2.2. Reduções β:

(a) $(\lambda x.x)a \rhd_\beta a$

(b) $(\lambda x.xxx)M \rhd_\beta MMM$

(c) $(\lambda x.(\lambda y.yx)z)v \rhd_\beta [v/x](\lambda y.yx)z \equiv (\lambda y.yv)z \rhd_\beta [z/y](yv) \equiv zv$

(d) $(\lambda x.xx)(\lambda x.xx) \rhd_\beta [(\lambda x.xx)/x]xx \equiv_\alpha (\lambda x.xx)(\lambda x.xx) \rhd_\beta [(\lambda x.xx)/x]xx \equiv_\alpha$
$(\lambda x.xx)(\lambda x.xx) \rhd_\beta [(\lambda x.xx)/x]xx \equiv_\alpha (\lambda x.xx)(\lambda x.xx) \rhd_\beta [(\lambda x.xx)/x]xx...$

É fácil constatar que a redução β é reflexiva e transitiva, mas não é simétrica pois dado um termo Q tal que $P \rhd_\beta Q$, a definição não fornece a construção reversa de P a partir de Q.

Definição 2.8. Forma Normal β

Um termo que não é, ou não tem, redex β é dito estar na forma normal β ou β-nf. Se um termo P reduz para Q na β-nf, Q é chamado de forma normal de P.

Observe que (d), também conhecido como o loop infinito Ω, sempre se mantém o mesmo, não importando quantas reduções sejam feitas. Como há redex e nunca reduz para β-nf, não existe forma normal para o termo.

Alguns termos podem ser reduzidos por mais de uma maneira. Por exemplo, (c) pode ser reduzido alternativamente como abaixo.

$$(\lambda x.(\lambda y.yx)z)v \rhd_\beta (\lambda x.[y](yx))v \equiv_\alpha (\lambda x.zx)v \rhd_\beta [v/x]zx \equiv_\alpha zv$$

Ambas as reduções levam a mesma forma normal, mas seria isto sempre verdade? Um sistema que se propõe a ser um modelo de computação precisa sempre apresentar uma mesma resposta independente do caminho de escolha durante o cálculo.

Alcançar a forma normal de um termo pode depender da ordem de avaliação. Por exemplo, o termo $(\lambda x.y)(\Omega)$ entra em loop com a avaliação do termo Ω mais à direita, porém resulta em y com a avaliação mais à esquerda.

Teorema 2.1. Teorema de Church-Rosser para a Redução β

Se $P \rhd_\beta M$ e $P \rhd_\beta N$, então existe um termo T (ver Figura 2.1) tal que

$$M \rhd_\beta T \text{ e } N \rhd_\beta T$$

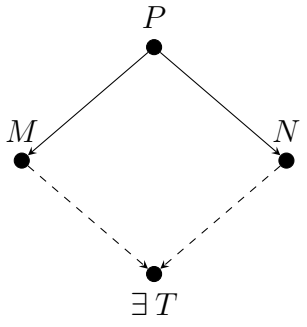

Figura 2.1: Teorema de Church-Rosser para a Redução β.

O teorema foi descoberto inicialmente por Church e Rosser, a prova original pode ser vista em (CHURCH; ROSSER, 1936). Uma versão padrão, a qual utiliza reduções paralelas pode ser vista em (TAKAHASHI, 1989). Diz-se que a redução β é confluente devido ao teorema. O principal resultado deste teorema é que uma computação, no fim, não terá dois resultados diferentes.

Corolário 2.1. Unicidade da Formal Normal β-nf

Se $P \rhd_\beta T_1$, $P \rhd_\beta T_2$, e T_1 e T_2 estão em β-nf , então $T_1 \equiv_\alpha T_2$.

Prova 2.1. Se $P \rhd_\beta T_1$, $P \rhd_\beta T_2$, e T_1 e T_2 estão em β-nf, pelo Teorema 2.1 T_1 e T_2 reduzem para T, mas ambos já estão em β-nf, logo $T_1 \equiv_\alpha T_2 \equiv_\alpha T$.

A equivalência entre termos pode ser estendida para além da questão sintática por meio da unicidade da forma normal. Se dois termos convergem para uma mesma forma normal, então eles são equivalentes. A unicidade de T é reforçada pela redução α, no sentido de que se M e N estiverem na forma β-nf, então $M \equiv_\alpha T$ e $N \equiv_\alpha T$.

Como o Cálculo Lambda é uma teoria sobre funções, necessita-se de uma método preciso que as compare. A teoria de equivalência α não consegue capturar todas as possíveis relações de igualdade entre os termos. Portanto, a proposta de uma relação de equivalência com base na redução β é mais interessante do que com a redução α. A redução β embora reflexiva e transitiva, é não-simétrica. Todavia é possível construir a seguinte estrutura relacional reflexiva, transitiva e simétrica.

Definição 2.9. Equivalência β

Um termo P é β equivalente a Q, relação denotada por $\equiv_\beta$, se existe um número finito de reduções $P_0, P_1, P_2 ... P_n (n \geq 0)$ tais que

$$(\forall i \leq \ n-1)(P_i \rhd_{1\beta} P_{i+1} \text{ ou } P_{i+1} \rhd_{1\beta} P_i \text{ ou } P_i \equiv_\alpha P_{i+1}), P_0 \equiv P, P_n \equiv Q$$

O que essa definição admite é que para qualquer i ou P_i é uma contração de P_{i+1}, ou o contrário, ou eles são *alpha* equivalentes. Dessa maneira, a propriedade de simetria é adicionada.

Com a definição de equivalência β é viável estender o teorema de Church-Rosser sobre $P \rhd_\beta T$ para o caso mais geral $P \equiv_\beta T$.

Teorema 2.2. Teorema de Church-Rosser para a Equivalência β

Se $P \equiv_\beta Q$, então existe um T tal que

$$P \rhd_\beta T \text{ e } Q \rhd_\beta T.$$

Prova: A prova acontece pela a indução do n na definição 2.9. Para $n = 0$ basta considerar $P_0 \equiv_\alpha P$ e $P_0 \equiv_\alpha Q$, logo pela transitividade da equivalência α tem-se $P \equiv_\alpha Q$. Portanto, se $P \rhd_\beta Q$ e $Q \rhd_\beta Q$ por uma redução β vazia, então $P \rhd_{1\beta} T$ e $Q \rhd_{1\beta} T$, onde $T \equiv_\alpha Q$. A indução do passo de n para $n+1$ acontece ao considerar-se como verdadeira a hipótese a ser testada, então deve existir um termo $P_n \equiv T'$, o qual $Q' \rhd_\beta T'$ e $P \rhd_\beta T'$. Sobretudo, o que se deseja provar é a existência de um termo T e não T', porém mostra-se que se existe um T', então T também existe. Considere que $P_{n+1} \equiv_\alpha Q$ ou $P_{n-1} \equiv_\alpha Q$, isto resulta em três possibilidades:

(a) $Q \equiv_\alpha Q'$, o que resulta em $T \equiv_\alpha T'$.

(b) $Q \rhd_\beta Q'$ quer dizer que Q antecede Q', logo $T \equiv_\alpha T'$ (ver Figura 2.2).

(c) $Q' \rhd_\beta Q$ quer dizer que Q' não 1β reduz a T, mas pelo teorema 2.1 deve existir um T tal que $T' \rhd_\beta T$ e $Q \rhd_\beta T$ (ver Figura 2.2).

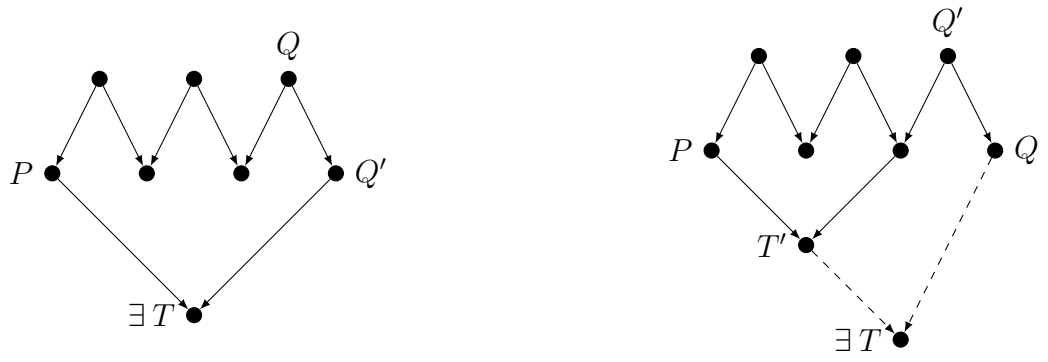

Figura 2.2: Prova do Teorema de Church-Rosser para Equivalência β. Na esquerda o caso (b) e na direita o caso (c)

O resultado importante deste teorema é que um termo P é β equivalente a no máximo um termo T em β-nf e para qualquer M, tal que $M \equiv_\alpha T$. A unicidade da forma normal β garante que se consideramos a forma normal como o significado de um termo, então ou o mesmo terá um único significado ou nenhum.

No Lambda Cálculo existe um tipo especial de abstração chamado de combinador de ponto fixo, pelo qual se encontra o ponto fixo de qualquer termo. O ponto fixo é um valor o qual passado a um operador não mudará de valor após qualquer quantidade de aplicações. Por exemplo, o ponto fixo da função $f(x) = 2x$ é 0, pois não alterará seu valor mesmo se for aplicado várias vezes: $f(f(f(0))) = 0$. Algumas funções matemáticas não tem ponto fixo como a função sucessora $s(x) = x + 1$. Outras têm mais de um ponto fixo, como a função de raiz quadrada $q(x) = \sqrt{x}$. A identidade $\lambda x.x$ tem todos os seus valores como ponto fixo.

Definição 2.10. Combinador

Um termo é um combinador se não tiver variáveis livres em seu escopo.

Formalmente, no Lambda Cálculo, o ponto fixo de um termo X é um termo P tal que

$$XP \equiv_\beta P.$$

Teorema 2.3. Teorema do Ponto Fixo

Existe um combinador Y capaz de encontrar o ponto fixo de qualquer termo X, de maneira que

$$YX \equiv_\beta X(YX)$$

Existem diversos combinadores de ponto fixo Y. Alan Turing descobriu o combinador $Y \equiv UU$, onde

$$U \equiv \lambda ux.x(uux)$$

e Haskell Curry descobriu o combinador $Y \equiv \lambda x.VV$, onde

$$V \equiv \lambda y.x(yy).$$

2.2 Cálculo Lambda Simplesmente Tipado

Na matemática a definição de uma função vem acompanhada de uma assinatura com o domínio e a contra-domínio que servem para dizer, respectivamente, quais são os seus conjuntos de entrada e saída. No Cálculo Lambda Atipado não existe restrição sobre a

entrada de uma função, desse modo, aplicações podem ocorrer de maneira que resultem em algo análogo a programas errados ou sem sentido. Por esse motivo, a grande maioria das linguagens contam com alguma forma de tipagem, pois tipos são úteis para especificar comportamentos básicos de funções e assim identificar erros em tempo de compilação.

Alonzo Church introduziu tipos no Cálculo Lambda pela primeira vez em 1940. Em seu modelo, tipos são rótulos atribuídos a termos para indicar seus conjuntos. Se um termo pertence a um conjunto σ, tem um tipo σ. Caso seja um elemento de um conjunto de funções de σ para α, então tem o tipo $\sigma \to \alpha$. O símbolo $\to$ é conhecido como um construtor de tipo. É plausível, numa teoria de tipos, outros construtores como o produto cartesiano $\times$, mas por simplicidade este modelo se limita apenas ao construtor de tipo funcional.

Na proposta de Church os tipos fariam parte da estrutura dos termos, portanto, termos, que caso violassem a definição não seriam considerados válidos dentro da teoria. Dessa maneira, o termo sem tipo $\lambda x.x$ não existe. Em contrapartida, para cada tipo σ existe uma variável x^σ e uma função $\lambda x^\sigma.x^\sigma$. Logo, para cada conjunto S, na teoria de Church, denotado pelo tipo σ, existe uma função identidade. A definição, ainda, restringe um único tipo por termo e a aplicação PQ é apenas definida quando P tem tipo $\sigma \to \tau$ e Q tem tipo σ.

Em 1958, Haskell Curry formulou um modelo de Cálculo Lambda Tipado similar ao de Church, porém ao invés de introduzir a assinatura de tipo na estrutura dos termos, os tipos seriam atribuídos a termos de acordo com um conjunto de regras de inferência. Em analogia, no modelo de Church, o tipo de um termo é como a impressão digital de uma pessoa, faz parte dela desde o seu nascimento. Já no modelo de Curry, o tipo de um termo é como um documento de identidade, a qual é atribuído após o nascimento (HINDLEY; SELDIN, 2008). A motivação de Curry era evitar que termos, como $\lambda x.x$, fossem descartados e que conceito intuitivo de identidade não fosse separado em casos especiais. O objetivo era formular uma teoria o qual existisse uma única função identidade $\lambda x.x$ e que tipos seriam atribuídos ela de acordo com um conjunto de regras formais.

Os teoremas de Church-Rosser (redução e equivalência β), destacados anteriormente, também são válidos para o Cálculo Lambda Tipado (Church e Curry). Entretanto, o mesmo não se pode dizer sobre o teorema do ponto fixo. Será apresentado uma demos-

tração de que os combinadores de ponto fixo não são tipáveis na Seção 2.2.2. Além disso, os teoremas apresentados para qualquer um dos modelos tipados são válidos para ambos, devido à equivalência dos estilos.

2.2.1 Estilo de Church

Este capítulo apresenta o Cálculo Lambda Simplesmente Tipado de Church, pois além de ser o primeiro modelo tipado, também é o mais simples.

Definição 2.11. Tipos Simples

Considere o conjunto finito $\mathbb{U}$ (ex: $\{c_1, c_2, c_3\}$) de símbolos chamados constantes de tipos, então $\mathbb{T}$ é o conjunto de todos os tipos e é definido como segue.

(i) Toda constante de tipo em $\mathbb{U}$ é um tipo em $\mathbb{T}$.

(ii) Se τ e τ' são tipos em $\mathbb{T}$, então a expressão $(\tau \to \tau')$ é um tipo em $\mathbb{T}$ e representa o tipo funcional (abstração λ) de τ para τ'.

Nota 2.2. Símbolos gregos em minúsculos τ, τ'... são ferramentas de meta-linguagem e não fazem parte da sintaxe de tipos.

A união de um tipo simples com uma variável é chamada de variável tipada, representada por $x^{\tau'}$, cujo significado é o elemento x pertence ao conjunto τ'. Como uma teoria sobre tipos não é igual a uma teoria sobre conjuntos, no sentido que existem restrições sobre quais elementos podem pertencer a quais conjuntos, então a seguinte definição se faz necessária.

Definição 2.12. Variáveis Tipadas

Dado um conjunto $\mathbb{V}$ de variáveis e um conjunto $\mathbb{T}$ de tipos simples,

(i) Se x é uma variável em $\mathbb{V}$ e τ é um tipo simples em $\mathbb{T}$, então $x^{\tau'}$ é uma variável tipada.

(ii) (Consistência) Nenhuma variável é tipada com mais de um tipo, ou seja, para $x^{\tau'}$ e x^{τ}, não é aceito que $\tau' \not\equiv \tau$.

(iii) Todos os tipos em $\mathbb{T}$ podem ser atribuídos a uma quantidade irrestrita de variáveis.

Tipos funcionais ($\rightarrow$) são associativos à direita, portanto $\tau \rightarrow ... \rightarrow \tau' \rightarrow \tau''$ denota $\tau \rightarrow (... \rightarrow (\tau' \rightarrow \tau''))$. As letras gregas são apenas nomes que seguram posições para tipos pertencentes ao conjunto $\mathbb{T}$. Exemplos válidos para τ e τ' são $\mathbb{N} \rightarrow \mathbb{N}$ e $\mathbb{B}$.

Na subseção anterior, apresentou-se como são termos λ bem formados, porém com inclusão de tipos é necessária uma reformulação de como termos podem ser combinados para formar novos termos.

Definição 2.13. Termos Simplesmente Tipados

(i) Todas as variáveis tipadas são termos λ tipados.

(ii) Se $M^{\tau \rightarrow \tau'}$ e N^{τ} são termos tipados, então $(M^{\tau \rightarrow \tau'} N^{\tau})^{\tau'}$ é um termo λ tipado.

(iii) Se x^{τ} é uma variável com tipo τ e $M^{\tau'}$ é um termo com tipo τ', então $(\lambda x^{\tau}.M^{\tau'})^{\tau \rightarrow \tau'}$ é um termo λ tipado.

Abaixo seguem alguns exemplos de termos tipados válidos.

(a) x^{τ}, para todo tipo τ em $\mathbb{T}$.

(b) $x^{\mathbb{N}}$

(c) $y^{\mathbb{N}}$

(d) $(M^{\mathbb{N} \rightarrow \mathbb{B}} x^{N})^{\mathbb{B}}$

(e) $(\lambda x^{\mathbb{N}}.x^{\mathbb{N}})^{\mathbb{N} \rightarrow \mathbb{N}}$

(f) $(\lambda z^{\rho}.((x^{\rho \rightarrow \tau \rightarrow \tau'} z^{\rho})^{\tau \rightarrow \tau'} (y^{\rho \rightarrow \tau} z^{\rho})^{\tau})^{\tau'})^{\rho \rightarrow \tau'}$ para todo τ, τ' e ρ em $\mathbb{T}$

Para verificar se os termos são válidos é necessário verificar se todos os tipos estão em concordância com a definição. O exemplo (f) deve ser analisado por partes. Primeiro, $(x^{\rho \rightarrow \tau \rightarrow \tau'} z^{\rho})^{\tau \rightarrow \tau'}$ e $(y^{\rho \rightarrow \tau} z^{\rho})^{\tau}$ estão de acordo com a definição de termo tipado ii, assim como $((x^{\rho \rightarrow \tau \rightarrow \tau'} z^{\rho})^{\tau \rightarrow \tau'} (y^{\rho \rightarrow \tau} z^{\rho})^{\tau})^{\tau'}$. Por fim, $(\lambda z^{\rho}.((x^{\rho \rightarrow \tau \rightarrow \tau'} z^{\rho})^{\tau \rightarrow \tau'} (y^{\rho \rightarrow \tau} z^{\rho})^{\tau})^{\tau'})^{\rho \rightarrow \tau'}$ está de acordo com a definição de termo tipado iii.

A definição de termo λ tipado normalmente é representada por meio de regras de inferência como segue, por ser mais visual apresentar verificações de tipo em diagramas em árvore.

$$\text{(i)} \quad x^{\tau'}$$

$$\text{(ii)} \quad \frac{M^{\tau \rightarrow \tau'} \qquad N^{\tau}}{(M^{\tau \rightarrow \tau'} N^{\tau})^{\tau'}}$$

$$\text{(iii)} \quad \frac{M^{\tau'}}{(\lambda x^{\tau}.M^{\tau'})^{\tau\to\tau'}}$$

A árvore de derivação do exemplo (f) é dada abaixo:

$$\dfrac{\dfrac{\dfrac{x^{\rho\to\tau\to\tau'} \qquad z^{\rho}}{(x^{\rho\to\tau\to\tau'}z^{\rho})^{\tau\to\tau'}} \qquad \dfrac{y^{\rho\to\tau} \qquad z^{\rho}}{(y^{\rho\to\tau}z^{\rho})^{\tau}}}{((x^{\rho\to\tau\to\tau'}z^{\rho})^{\tau\to\tau'}(y^{\rho\to\tau}z^{\rho})^{\tau})^{\tau'}}}{(\lambda z^{\rho}.((x^{\rho\to\tau\to\tau'}z^{\rho})^{\tau\to\tau'}(y^{\rho\to\tau}z^{\rho})^{\tau})^{\tau'})^{\rho\to\tau'}}$$

A nova definição delimita consideravelmente quais termos podem ser definidos. O termo que generaliza a auto aplicação, $\lambda x^{\tau'}.x^{\tau'}x^{\tau'}$, não pode existir, pois conforme a definição (ii) a aplicação xx necessita que o primeiro x seja um termo do tipo $\tau \to \tau'$ e o segundo x precisa ter o tipo τ. Contudo, como estabelecido pela condição de consistência, uma mesma variável não pode ser associada a mais de um tipo num mesmo contexto.

Definição 2.14. Substituição Tipada

A substituição tipada acontece pela troca de todas ocorrências, em $M^{\tau'}$, de x^{τ} por N^{τ} e é denotada por $[N^{\tau}/x^{\tau}](M^{\tau'})$. Observe que não se define a substituição de N e x com tipos diferentes, por exemplo $[N^{\tau}/x^{\tau'}](M^{\tau'})$.

O processo de substituição apresentado na subseção anterior é mantido aqui. Uma questão relevante é se o termo resultante após uma substituição continua sendo um termo válido no Cálculo Lambda Simplesmente Tipado. Realmente, é possível provar que dado um termo tipado $M^{\tau'}$, a substituição das ocorrências de P^{τ} por outro termo com tipo τ resultará em M com tipo τ'.

A Redução τ utiliza a substituição e tem, também, garantida a preservação dos tipos. Assim, a conversão $(\lambda y^{\tau}.[y^{\tau}/x^{\tau}]M^{\tau'})^{\tau\to\tau'}$ resulta num termo com o mesmo tipo que $(\lambda x^{\tau}.M^{\tau'})^{\tau\to\tau'}$. A Redução β preserva igualmente os tipos. A aplicação $[N^{\tau}/x^{\tau}]M^{\tau'}$ resulta num termo com mesmo tipo que $((\lambda x^{\tau}.M^{\tau'})^{\tau\to\tau'}N^{\tau})^{\tau'}$. A prova desses lemas é irrelevante para a conclusão deste trabalho.

2.2.2 Estilo de Curry

Como foi supracitado, o estilo de Church divide cada função em casos especiais ao definir o tipo como parte da definição de termo λ. Uma função com tipo τ existe para cada τ no conjunto de tipos $\mathbb{T}$. Curry tomou uma abordagem diferente, pois acreditava que

uma operação deveria ser representada por um único termo e, então, tipos poderiam ser atribuídos mediante a disciplina de tipos.

O estilo de Curry, também conhecido como *Type-Assignment* (TA), utiliza regras de inferência para atribuir tipos aos termos, similares às do sistema Dedução Natural de Gentzen. Esta relação entre lógica e sistemas de tipos é conhecida como Isomorfismo de Curry-Howard.

Para que uma função pudesse ser associada a um único tipo τ que representasse todas as suas possíveis instâncias, Curry introduziu variáveis de tipo na definição do conjunto de tipos $\mathbb{T}$. Esse mesmo conceito está presente em diversas linguagens de programação, inclusive Haskell, por meio do polimorfismo paramétrico, que possibilita a escrita de programas mais genéricos.

A definição de termo é igual à do Cálculo Lambda Atipado (Definição 2.1), porém adiciona-se regras para tuplas de termos, pois será útil para construção do triplo isomorfismo Curry-Howard-Lambek da Seção 5.

Definição 2.15. Termo λ

(iii) O símbolo $*$ é um termo λ.

(iv) Se M e N são termos λ, então $\langle M, N\rangle$ é um termo λ (produto).

(v) Se M é um termo λ, então $\pi_1(M)$ é um termo λ (projeção).

(vi) Se M é um termo λ, então $\pi_2(M)$ é um termo λ (projeção).

Esta modificação sintática é apenas por praticidade, não torna o formalismo mais poderoso e nem menos. Poderia-se representá-la como termos λ como segue:

$$\langle x, y\rangle \equiv (\lambda xyz.zxy);$$
$$\pi_1(p) \equiv (\lambda p.p(\lambda xy.x));$$
$$\pi_2(p) \equiv (\lambda p.p(\lambda xy.y)),$$

onde as funções representam, respectivamente, a construção de uma tupla e a extração do primeiro e do segundo elemento de uma tupla. A substituição precisa ser estendida conforme:

Definição 2.16. Regras de Substituição:

(g) $[N/x]\langle P, Q\rangle \equiv \langle [N/x]P, [N/x]Q\rangle$;

(h) $[N/x]\pi_1(P) \equiv \pi_1([N/x]P)$;

(i) $[N/x]\pi_2(P) \equiv \pi_2([N/x]P)$;

O conjunto FV de variáveis livres é estendido conforme:

$$FV(\pi_1(M)) = FV(M)$$
$$FV(\pi_2(M)) = FV(M)$$
$$FV(\langle M, N\rangle) = FV(M) \cup FV(N)$$

A Redução α (Definição 2.5) não precisa ser modificada, visto que a nova definição de substituição é suficiente para estabelecer

$$\lambda x.\langle P, Q\rangle \rightarrow_\alpha \lambda y.\langle [y/x]P, [y/x]Q\rangle,\ se\ y \notin FV(\langle P, Q\rangle)$$
$$\lambda x.\pi_1(P) \rightarrow_\alpha \lambda y.\pi_1([y/x]P),\ se\ y \notin FV(\pi_1(P))$$
$$\lambda x.\pi_2(P) \rightarrow_\alpha \lambda y.\pi_2([y/x]P),\ se\ y \notin FV(\pi_2(P))$$

Por outro lado, a redução β precisa ser modificada para adicionar significado às funções π_1, π_2, como segue:

Definição 2.17. Redução β

$$(\lambda x.M)N \rhd_{1\beta} [N/x]M$$
$$\pi_1(\langle M, N\rangle) \rhd_{1\beta} M$$
$$\pi_2(\langle M, N\rangle) \rhd_{1\beta} N$$

Nota 2.3. Por conveniência adota-se, para o restante da monografia, termo como sinônimo de termo λ.

Um termo X, em TA, recebe um tipo τ se, e somente se, existe um termo X^τ no modelo de Church (excetua-se tuplas). No entanto, a definição de tipo é consideravelmente diferente e existe um mecanismo, mais elaborado para atribuir tipos a termos, conhecido como Sistema de Atribuição de Tipo (*Type-Assignment System*).

Definição 2.18. Tipos Paramétricos

Considere o conjunto infinito de variáveis de tipo $\mathbb{W}$ (ex: $\{a, b, c...\}$) e o conjunto finito de constantes de tipo $\mathbb{C}$ (ex: $\{c_1, c_2, c_3\}$). Assim, $\mathbb{T}_p$ é o conjunto de todos os tipos (paramétricos) e é definido como.

(i) 1 é um tipo constante e pertence à $\mathbb{T}_p$.

(ii) Toda variável de tipo em $\mathbb{W}$ e toda constante de tipo em $\mathbb{C}$ são tipos em $\mathbb{T}_p$.

(iii) Se τ e τ' são tipos em $\mathbb{T}_p$, então a expressão $(\tau \to \tau')$ é um tipo em $\mathbb{T}_p$ e representa o tipo funcional (abstração λ) de τ para τ'.

(iv) Se τ e τ' são tipos em $\mathbb{T}_p$, então a expressão $(\tau \times \tau')$ é um tipo em $\mathbb{T}_p$ e representa o tipo produto cartesiano entre τ e τ'.

A maneira com que os tipos são associados às variáveis não é mais parte da sintaxe, por esse motivo, existe uma alteração na notação. Tipos passam a ser propriedades de termos, podendo ou não as ter.

Definição 2.19. Atribuição de Tipo

Dado um tipo paramétrico τ e um termo M, uma atribuição de tipo é denotada por

$$M : \tau,$$

cujo significado é o termo M tem o tipo τ.

Definição 2.20. Contexto

Um contexto é um conjunto Γ da forma $\{x_1 : \tau_1, x_2 : \tau_2, x_3 : \tau_3...\}$ de atribuições de tipo, cujos termos são variáveis de $\mathbb{V}$ e as seguintes condições são seguidas:

(i) (Consistência) Nenhuma variável é atribuída a mais de um tipo, ou seja, para $x : \tau$ e $x : \tau'$, não se aceita que $\tau \not\equiv \tau'$.

(ii) Todo tipo τ pode ser atribuído a uma quantidade irrestrita de variáveis.

A definição de contexto respeita as mesmas condições da Definição 2.12. O contexto normal é apresentado como conjunto de suposições, as quais ainda não foram

provadas serem verdades e esse é o seu significado durante o processo de verificação do tipo de um termo. Nem sempre o contexto é feito explícito em sistemas *a la* Curry, como em (HINDLEY; SELDIN, 2008). Por conseguinte, é necessário manter em mente quais suposições ainda não foram descarregadas (provadas). Provar que uma suposição $x : \tau$ é verdadeira, ou descarregá-la, é o procedimento de remover uma atribuição de tipo $x : \tau$ de um contexto Γ e é denotado por $\Gamma \setminus x$. Note que se $x \notin \Gamma$, então $\Gamma \setminus x = \Gamma$.

Os contextos Γ_1 e Γ_2 serão consistentes entre si se $\Gamma_1 \cup \Gamma_2$ também for. Se a união de $\Gamma_1, \Gamma_2, \Gamma_3, ..., \Gamma_n$ é consistente, diz-se que são mutualmente consistentes.

Definição 2.21. Fórmula TA

Dado um contexto Γ, um termo M e um tipo paramétrico τ, a tripla $\langle \Gamma, M, \tau \rangle$ é chamada de Fórmula TA e é denotada por

$$\Gamma \vdash M : \tau.$$

E afirma-se que M tem tipo τ com o contexto Γ.

Definição 2.22. Regras do Sistema *Type-Assignment System*

Deduções sem pré-condição:

$$\frac{}{x : \tau \vdash x : \tau}\,(var) \qquad \frac{}{\vdash * : 1}\,(unit)$$

Regras de dedução (Aplicáveis somente se os contextos forem consistentes)[1]: .

$$\frac{\Gamma_1 \vdash M : \tau \to \tau' \qquad \Gamma_2 \vdash N : \tau}{\Gamma_1 \cup \Gamma_2 \vdash (MN) : \tau'}\,(app) \qquad \frac{\Gamma \vdash M : \tau'}{\Gamma \setminus x : \tau \vdash (\lambda x.M) : \tau \to \tau'}\,(abs)$$

$$\frac{\Gamma_1 \vdash M : \tau \qquad \Gamma_2 \vdash N : \tau'}{\Gamma_1 \cup \Gamma_2 \vdash \langle M, N \rangle : \tau \times \tau'}\,(pair) \qquad \frac{\Gamma \vdash M : \tau \times \tau'}{\Gamma \vdash \pi_1(M) : \tau}\,(\pi_1) \qquad \frac{\Gamma \vdash M : \tau \times \tau'}{\Gamma \vdash \pi_2(M) : \tau'}\,(\pi_2)$$

O sistema é similar ao da Definição 2.13, porém a introdução de um termo por (var) é dado como uma dedução e existe uma regra especial $(unit)$ para atribuir unicamente o tipo 1 ao termo $*$. A regra de dedução (abs) ou $x : \alpha \in \Gamma$ ou Γ para qualquer tipo α. O primeiro caso é conhecido como descarregamento (ou prova) de $x : \alpha$. O segundo é um descarregamento vazio, simbolizado por (abs_v).

[1]Para todo o resto desta monografia esta condição é considerada

Definição 2.23. Dedução e Prova em TA

Uma dedução em TA é uma árvore, cujos topos são axiomas e as demais partes inferiores são regras de dedução. A fórmula mais inferior é chamada de conclusão.

Se o contexto da conclusão for o conjunto vazio $\emptyset$, então, afirma-se que a dedução é uma prova. Uma atribuição de tipo $M : \tau$ provada em TA é representada por $TA \vDash M : \tau$.

A função identidade ($I \equiv \lambda x.x$) pode ser tipada para algum tipo a em $\mathbb{V}$ pela dedução do Exemplo 2.3.a, portanto ela existe uma única vez em TA. Percebe-se o uso de descarregamentos vazios na dedução de Exemplo 2.3.b. A dedução do Exemplo 2.3.c é um exemplo mais complexo de uma abstração que faz a composição de outros dois termos, também conhecido como combinador B.

Exemplo 2.3. Provas em TA

$$\text{(a)} \quad \dfrac{x : a \vdash x : a}{\vdash (\lambda x.x) : a \to a}\,(abs)$$

$$\text{(b)} \quad \dfrac{\dfrac{\dfrac{y : b \vdash y : b}{y : b \vdash (\lambda z.y) : a \to b}\,(abs_v)}{\vdash (\lambda yz.y) : b \to a \to b}\,(abs)}{\vdash (\lambda xyz.y) : c \to b \to a \to b}\,(abs_v)$$

$$\text{(c)} \quad \dfrac{\dfrac{\dfrac{\dfrac{x : a \to b \vdash x : a \to b \qquad \dfrac{y : c \to a \vdash y : c \to a \qquad z : c \vdash c}{y : c \to a, z : c \vdash yz : a}\,(app)}{y : c \to a, z : c, x : a \to b \vdash x(yz) : b}\,(app)}{y : c \to a, x : a \to b \vdash (\lambda z.x(yz)) : c \to b}\,(abs)}{x : a \to b \vdash (\lambda yz.x(yz)) : (c \to a) \to c \to b}\,(abs)}{\vdash (\lambda xyz.x(yz)) : (a \to b) \to (c \to a) \to c \to b}\,(abs)$$

$$\text{(d)} \quad \dfrac{\dfrac{x : a \to a \vdash x : a \to a}{\vdash (\lambda x.x) : (a \to a) \to (a \to a)}\,(abs) \qquad \dfrac{x : a \vdash x : a}{\vdash (\lambda x.x) : a \to a}\,(abs)}{\vdash (\lambda x.x)(\lambda x.x) : a \to a}\,(app)$$

$$\text{(e)} \quad \dfrac{\dfrac{\dfrac{x : a \vdash x : a \qquad y : b \vdash y : b}{x : a, y : b \vdash \langle x, y\rangle : a \times a}\,(pair)}{x : a \vdash \lambda y.\langle x, y\rangle : b \to a \times b}\,(abs)}{\vdash \lambda yx.\langle x, y\rangle : b \to a \to a \times b}\,(abs)$$

O termo da autoaplicação geral $\lambda x.xx$ não existe no modelo de Church, mas no sistema TA existe. Porém não é possível atribuir um tipo pois é necessário assumir x ter dois tipos diferentes em um mesmo contexto (veja a dedução (2.4a)), logo o contexto

resultante de ambas suposições é inconsistente. Surpreendentemente, a autoaplicação da identidade II pode ser tipada, conforme mostra a dedução (2.3d). Observa-se que a eliminação da autoaplicação exclui todos os casos sem terminação como Ω, pois se $TA \nvdash (\lambda x.xx) \Rightarrow TA \nvdash \Omega$.

Exemplo 2.4. Contraexemplo de Dedução em TA

$$\text{(a)} \quad \dfrac{\dfrac{x : a \to c \vdash x : a \to c \qquad x : a \vdash x : a}{\Gamma \vdash xx : c}\,(app)}{\Gamma \setminus x \vdash (\lambda x.xx) : b \to c}\,(abs)$$

Definição 2.24. Termo Fortemente e Fracamente Normalizável

Tanto no Cálculo Lambda Atipado quanto no Tipado, um termo é fracamente normalizável (*weakly normalizable*, ou WN) se tem uma forma normal $\beta - nf$. Um termo é fortemente normalizável (*strong normalizable*, ou SN) se todas as possíveis reduções têm tamanho finito.

Evidentemente, um termo SN também é WN. O Termo $(\lambda x.y)\Omega$ tem uma forma normal se avaliado mais à esquerda, portanto é WN. Já o termo Ω não tem forma normal, e $(\lambda x.y)z$ tem todas as suas possíveis reduções num número finito de passos, então é SN.

Um termo fortemente normalizável pode ser visto como seguro de ser aplicado a reduções β. Em certo sentido, é um programa que não pode entrar em loop infinito. Um termo fracamente normalizável pode ser reduzido à forma normal, mas depende da ordem que os seus subtermos são reduzidos.

Como já reportado, tipos restringem consideravelmente os termos que existem. Neste modelo tipado, não só o termo geral da autoaplicação deixa de existir, como qualquer termo que não seja fortemente normalizável. O teorema abaixo estabelece isso.

Teorema 2.4. Cálculo Lambda Tipado é Fortemente Normalizável

Todos os termos do Cálculo Lambda Tipado são fortemente normalizáveis, todas as reduções β são finitas. A prova pode ser encontrada em (SØRENSEN; URZYCZYN, 1998).

2.3 Tipo Principal

As provas do Exemplo 2.3, para um dado termo, resultam numa assinatura de tipo ou em alguma inconsistência. Poderia-se questionar se a assinatura de tipo gerada por uma prova é que melhor representa o termo, no sentido de que existem outras assinaturas de tipo que são apenas instâncias obtidas por substituições de tipos. O tipo que melhor representa um termo é o seu Tipo Principal (Principal Type, ou PT).

Encontrar o tipo principal de um termo é um procedimento que deve, implicitamente, responder sim ou não para o problema de um termo ser tipável.

Definição 2.25. Termo Tipável

Um termo M é dito tipável se existe um tipo τ, tal que

$$\vDash M : \tau.$$

Decidir se um termo é tipável está relacionado com o problema da habitabilidade de um tipo. Porém, é uma inversão da busca, pois a questão passa a ser encontrar algum termo que satisfaça o tipo fornecido.

Definição 2.26. Tipo Habitável

Um tipo τ é dito habitável se existe um termo M, tal que

$$\vDash M : \tau.$$

Nota 2.4. O problema de verificar se um tipo é habitável é conhecido como *inhabitation*.

A definição de tipo principal depende da definição de substituição de tipo, pois é como instâncias são encontradas.

Definição 2.27. Substituição de Tipo

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

$$[\sigma_1/a_1, \sigma_2/a_2, \sigma_3/a_3, ..., \sigma_n/a_n],$$

onde σ_i são quaisquer tipos e a_i são distintas variáveis de tipo de τ. A aplicação de $\mathbb{S}$ em τ é denotada por $\mathbb{S}(\tau)$ e definida por

$$\begin{aligned} \mathbb{S}(a_i) &\equiv \sigma_i, \\ \mathbb{S}(b) &\equiv b \; se \; b \; \notin \{a_1, a_2, a_3, ..., a_n\}, \\ \mathbb{S}(\alpha \to \rho) &\equiv \mathbb{S}(\alpha) \to \mathbb{S}(\rho). \end{aligned}$$

Nota 2.5. Uma substituição onde $\{\sigma_1, \sigma_2, \sigma_3, ..., \sigma_n\}$ são variáveis distintas é chamada de renomeação de tipo.

Nota 2.6. Uma substituição de tipos pode ser aplicada a um contexto, por meio da aplicação da substituição no tipo de cada variável.

Definição 2.28. Contexto das variáveis livres de um termo

Se M é um termo, $FV(M) - context$ é um contexto cujas variáveis são exatamente as que ocorrem livres em M.

O tipo $\mathbb{S}(\tau)$ é uma instância de τ e propicia a seguinte definição de Tipo Principal em TA.

Definição 2.29. Tipo Principal

Seja M qualquer termo, com $FV(M) = \{x_1, x_2, ..., x_n\}$ $(n \geq 0)$.

(i) Se $n = 0$, então τ é o tipo principal de M se, e somente se, $\Gamma \vdash M : \tau'$ para algum Γ e para algum τ' que pode ser obtido por alguma substituição de tipos aplicada a τ.

(ii) Se $n \geq 0$: chama-se $\langle \Gamma, \tau \rangle$ de par-principal e τ de tipo principal de M, se, e somente se, Γ é $FV(M) - context$ e a relação $\Gamma' \vdash M : \tau'$ é válida para um $FV(M) - context$ Γ' e um tipo τ' somente quanto o par $\langle \Gamma', \tau' \rangle$ pode ser obtido por alguma mesma substituição de tipos aplicada a cada elemento do par principal $\langle \Gamma, \tau \rangle$.

Corolário 2.2. Unicidade do Tipo Principal

Seja τ o tipo principal de M, então τ é único salvo por renomeações de tipo. Portanto, qualquer outro tipo σ é um Tipo Principal de M se, e somente se, σ é obtido por uma renomeação de τ.

Do ponto de vista de linguagens de programação, encontrar o Tipo Principal de um programa é útil para a generalização de programas. Logo, um algoritmo que decida o PT de um programa é importante na prática. O algoritmo deve decidir se um programa é tipável e caso sim, retornar o Tipo Principal. Outra consequência é que qualquer erro de tipo pode ser identificado em tempo de compilação, como dito no Capítulo 3.

Nota 2.7. Inferência de Tipo é outro nomenclatura, geralmente utilizada em linguagens de programação, para o processo de encontrar o Tipo Principal.

Os algoritmos de inferência remetem a ideias da lógica proposicional, em particular à regra D (*Rule D*, ou *Condensed Detachment*) que foi, talvez, descrita informalmente por Alfred Tarski, em torno de 1930[1]. Haskell Curry descreve, informalmente, possivelmente o primeiro algoritmo de inferência para Lógica Combinatória em 1958 (CURRY; FEYS, 1958), que utiliza um método de solução equacional para os tipos. Posteriormente, Curry reescreve seu método e adiciona uma prova de sua corretude em (CURRY, 1969). O lógico Hindley (HINDLEY, 1969) também publica um algoritmo junto de sua comprovação, porém difere do método de Curry por utilizar o procedimento de unificação de Robison. A unificação é o procedimento de encontrar uma substituição $\mathbb{S}$ como descrito abaixo.

Definição 2.30. Tipos Unificáveis

Dois tipos τ e τ' são unificáveis se existe uma substituição $\mathbb{S}$ que os torna iguais, ou seja,

$$\mathbb{S}(\tau) = \mathbb{S}(\tau').$$

Tal substituição $\mathbb{S}$ é chamada de unificador dos tipos τ e τ'. O unificador mais geral é uma substituição $\mathbb{S}_g$ se, para qualquer outro unificador $\mathbb{S}$, existe uma outra substituição $\mathbb{S}'$, tal que

$$\mathbb{S}' \circ \mathbb{S} = \mathbb{S}_g.$$

Provavelmente o algoritmo de inferência mais popular é o Algoritmo W do Sistema Damas-Milner, que foi implementado na linguagem de programação ML (MILNER, 1978) e atualmente serve como principal referência para pesquisas em sistema de tipos. O Sistema Damas-Milner, em essência, é o mesmo utilizado por Hindley em 1969, entretanto, ao invés de Lógica Combinatória é aplicado a uma variante do Cálculo Lambda com operador *let*. Hindley apresenta uma versão do Algoritmo W para o sistema *Type Assignment* da Seção 2.2.2 em (HINDLEY, 1997) [1].

[1]Mencionado em (KALMAN, 1983)

[1]O autor desta monografia adicionou uma implementação em Haskell deste algoritmo no Apêndice A.

3 Sistemas de Tipos

Sistemas de tipos são regras que ditam quais expressões podem ser associadas a quais tipos. Em sistemas formais, como o Cálculo Lambda, servem para evitar paradoxos e computações infinitas. Nas primeiras linguagens de programação, sistemas de tipos apenas classificavam os dados para serem tratados corretamente pelo processador, garantindo o sentido operacional do programa. Com o avanço das linguagens, passaram a ser utilizados como regras para identificar falhas na consistência de programas. Este tipo de verificação expõem não só falhas triviais na lógica do programador, mas também profundos erros conceituais (PIERCE, 2002).

As regras do sistema de tipo são utilizadas por um algoritmo de verificação, que averígua se o tipo atribuído a um termo/função é válido (SEBESTA, 2012). A verificação acontece ou em tempo de compilação (*Statically Typed*) ou em tempo de execução (*Dynamically Typed*), ou em ambos. O primeiro pode ser pensado como mais seguro contra falhas, pois muitos erros são identificados antes da execução. (MITCHELL, 1996) sumariza três principais vantagens da verificação em tempo de compilação:

- Detecção antecipada de erros: erros como somar uma *string* a um inteiro podem ser detectados pelo verificador em tempo de compilação, ou seja, antes da execução.
- Documentação: assinatura de tipo de uma função pode servir como uma série de indícios para entender o seu funcionamento e seu uso.
- Garantia de validade de otimizações: ao acessar uma *array* de um tipo de dado, se o tamanho do dado é conhecido, então o compilador pode determinar uma função de acesso mais eficiente. Isto acontece na compilação de *records* do Pascal e *structs* do C.

Verificação em tempo de compilação é uma abordagem conservadora, pois além de descartar todos os programas com erros de tipo, também rejeita possíveis programas corretos. Idealmente, todas linguagens deveriam rejeitar programas incorretos ou, então, tratar o erro de maneira lógica em tempo de execução. Infelizmente esse não é o caso, por exemplo o código abaixo do JavaScript é aceito e é avaliado de maneira incoerente.

```
[] + 1; // "1"
```

Falhas como esta em sistema de tipo podem estar relacionadas com ausência de uma especificação formal e matemática do mesmo, o que pode resultar em inconsistências e pouca confiabilidade na corretude do programa. No caso de JavaScript há uma especificação textual da semântica de coerção e o comportamento estranho é uma escolha de projeto.

O sistema de tipos conhecido como Damas-Milner, ou Hindley-Milner (HM), usado em linguagens como ML, Miranda e Haskell, é capaz de identificar o tipo de todas as possíveis expressões. Um termo bem tipado em HM nunca causa uma computação equivocada (por exemplo, somar um inteiro com uma lista), o que é uma garantia de segurança, pois nenhum um erro de tipo acontece em tempo de execução. O sistema HM é fruto direto dos avanços da lógica no século passado, como a criação da Teoria dos Tipos e do Cálculo Lambda.

Este capítulo tratará de sistemas de tipos de linguagens de programação, em especial Haskell. Para este capítulo opta-se por algumas mudanças em como as definições sintáticas são feitas, que até então foram apresentadas por meio de indução de conjuntos, porém Gramáticas na Forma Normal de Backus são mais concisas e também mais frequentes na Ciência da Computação.

3.1 Sistema Damas-Milner

O Sistema Damas-Milner (DM) é um Cálculo Lambda Tipado bastante similar ao TA, utiliza quase a mesma definição de Termo λ e tipos também são atribuídos conforme o estilo de Curry. Foi criado pelo cientista da computação Robin Milner (MILNER, 1978) para linguagem funcional ML que é uma linguagem de programação de uso geral, mas que inicialmente era apenas uma metalinguagem para o assistente de provas *Logic of Computable Functions* (LCF). Segundo (DAMAS; MILNER, 1982), a linguagem apresentou-se bem-sucedida ao longo dos anos, devido à combinação de flexibilidade (polimorfismo), robustez (expressividade semântica ou *semantic soundness*) e detecção de erros em tempo de compilação.

Luis Damas, aluno de Milner, provou que o sistema era completo: no sentido de que todo tipo derivável é uma instância do tipo computado pelo Algoritmo W (DAMAS,

1985). Como afirmado na Seção anterior, Damas criou o mesmo sistema de Hindley sem saber que estava, na realidade, redescobrindo [1] um sistema criado uma década atrás.

O operador *let* está presente em diversas linguagens de programação funcionais desde sua introdução na linguagem do assistente de provas LCF, desenvolvida por Milner, Gordon e Wadsworth com base no trabalho não publicado de Dana Scott. O objetivo do *let* é permitir definições locais, de maneira que uma variável atue como um nome para um Termo λ. Por exemplo,

$$let\, id \;=\; \lambda\, x\, .xin\, id.$$

Termo λ, da Definição 2.1, é estendida como segue:

Definição 3.1. Termo λ

Na gramática abaixo, **e** representa uma expressão e **x** uma variável qualquer.

$$e ::= x \mid ee' \mid \langle e, e' \rangle \mid \lambda x.e \mid let\, x \;=\; e\, in\, e'$$

Os tipos em DM são uma extensão do TA com a adição do quantificador lógico $\forall$, o que significa que variáveis de tipo agora podem ser livres ou ligadas. Os tipos em TA, da Definição 2.18 de Tipos Paramétricos, são chamados de Monotipos (Monotypes) em DM. Politipo (*Politypes ou Type-Scheme*) é definido como segue:

Definição 3.2. Politipo

Seja a gramática:

$$\tau ::= \alpha \mid c \mid \tau \rightarrow \tau' \mid \tau \times \tau'$$
$$\sigma ::= \tau \mid \forall \alpha.\sigma,$$

onde α é uma variável de tipo, c é uma constante de tipo, τ é um monotipo e σ é um politipo.

A sintaxe de tipos ficou mais complexa, pois existem dois conjuntos de tipo: um sem quantificador $\forall$ e outro com, que pela regra sintática apenas pode acontecer à esquerda de um monotipo ou outro quantificador. As definições de atribuição de tipos, de contexto e de fórmula (Definições de 2.19 até 2.21) da Seção 2.2.2 estão presentes em DM

[1] É possível discutir se foi uma redescoberta ou uma coincidência ambos criarem exatamente o mesmo sistema.

com as devidas adaptações. Atribuição de tipo e Fórmula são redefinidas para politipos ao invés de Tipo Paramétrico, com a devida troca de palavras. Quando é trivial assim, apresenta-se uma nova definição com o mesmo nome, como por exemplo a definição de variáveis de tipo livres, ligadas e ligantes.

Definição 3.3. Variáveis de tipo livres, ligadas e ligantes

Um variável de tipo α é

(i) ligada se ocorre em σ no tipo $\forall\alpha.\sigma$.

(ii) ligante caso $\forall\alpha.\sigma$.

(iii) livre caso não seja ligada e nem ligante.

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

$$
\begin{aligned}
&FTV(\alpha) = \{\alpha\} \\
&FTV(\tau \to \tau') = FTV(\tau) \cup FTV(\tau') \\
&FTV(\tau \times \tau') = FTV(\tau) \cup FTV(\tau') \\
&FTV(\forall\alpha.\sigma) = FTV(\sigma) \setminus \{\alpha\}
\end{aligned}
$$

A principal motivação do operador $\forall$ é criar uma relação de especialização denotada por $\alpha > \tau$, significando que τ é mais especializado que α. A especialização (ou instanciação) de tipo é uma substituição $\mathbb{S}$ aplicada a um politipo, logo é necessário reformular a substituição de tipos para considerar tipos quantificados.

Definição 3.4. Substituição de Tipo

Uma substituição de tipo $\mathbb{S}$ em um politipo τ é uma sequência,

$$[\sigma_1/a_1, \sigma_2/a_2, \sigma_3/a_3, ..., \sigma_n/a_n],$$

onde σ_i são quaisquer politipos e a_i são distintas variáveis de tipo de τ. A aplicação de $\mathbb{S}$, da Definição 2.27, é estendida por

$$\mathbb{S}(\forall\alpha.\sigma) \equiv \forall\mathbb{S}(\alpha).\mathbb{S}(\sigma).$$

Além de substituir tipos, a especialização opcionalmente elimina quantificadores, por exemplo:

$$\forall a.a \to a > a \to a$$

Nota-se que σ_i, em $\mathbb{S}$, pode ser um monotipo ou um politipo. Suponha a aplicação de uma substituição $\mathbb{S}(\tau)$. No primeiro caso, existe a possibilidade de quantificar uma variável de tipo livre, como em:

$$[a/b]\forall a.b \to a \to a \equiv \forall a.a \to a \to a,$$

o que iria de encontro com a noção de mais especial. Assim, só deve ser permitido fazer a substituição de tipos em variáveis de tipo que estão ligadas a um quantificador. Já no segundo caso, se τ for um politipo, pode ocorrer a quebra da Definição 3.2 (sintática) de politipo e a quantificação de variáveis que eram livres no tipo original, por exemplo:

$$[\forall\beta.\beta \to \beta/\alpha]\alpha \to \alpha \equiv (\forall\beta.\beta \to \beta) \to (\forall\beta.\beta \to \beta).$$

A única opção é limitar a substituição, na especialização, para apenas monotipos. Abaixo há uma regra que generaliza o caso de τ ser um monotipo e que também permite a relação

$$\forall\alpha.\alpha \to \alpha > \forall\beta.\beta \to \beta > \forall\alpha.\alpha \to \alpha,$$

que mostra um caso especial de simetria entre os tipos ao renomear as variáveis quantificadas.

A leitura é feita no sentido horário: num politipo $\forall\alpha_1...\alpha_n.\tau$, aplique a substituição $[\sigma_i/\alpha_i]$[1] em τ e assim obtenha o monotipo τ'. Em seguida, as variáveis de tipos β_i não livres no tipo original são, opcionalmente, quantificadas.

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

É possível demostrar que essa regra estabelece $>$ como uma relação reflexiva, transitiva e antissimétrica. Pois para quaisquer tipos α, σ e ρ

$$\sigma > \sigma \text{ (reflexividade)},$$

$$\sigma > \alpha, \alpha > \rho \Rightarrow \sigma > \rho \text{ (transitividade)},$$

$$\sigma > \alpha, \alpha > \sigma \Rightarrow \alpha \equiv {}^2\sigma \text{ (antissimétrica)}.$$

[1]Note que α_i são variáveis de tipos ligadas e que σ_i são monotipos.

[2]A equivalência $\equiv$ sintática é garantida por alguma renomeação das variáveis de tipo em α.

Definição 3.5. Regras do Sistema Damas-Milner

$$\frac{}{x : \tau \vdash x : \tau}\,(var) \qquad \frac{}{\vdash * : 1}\,(unit)$$

$$\frac{\Gamma_1 \vdash M : \tau \to \tau' \qquad \Gamma_2 \vdash N : \tau}{\Gamma_1 \cup \Gamma_2 \vdash (MN) : \tau'}\,(app) \qquad \frac{\Gamma \vdash M : \tau'}{\Gamma \setminus x : \tau \vdash (\lambda x.M) : \tau \to \tau'}\,(abs)$$

$$\frac{\Gamma_1 \vdash M : \tau \qquad \Gamma_2 \vdash N : \tau'}{\Gamma_1 \cup \Gamma_2 \vdash \langle M, N\rangle : \tau \times \tau'}\,(pair) \qquad \frac{\Gamma \vdash M : \tau \times \tau'}{\Gamma \vdash \pi_1(M) : \tau}\,(\pi_1) \qquad \frac{\Gamma \vdash M : \tau \times \tau'}{\Gamma \vdash \pi_2(M) : \tau'}\,(\pi_2)$$

$$\frac{\Gamma_1 \vdash M : \sigma \qquad \Gamma_2 \cup \{x : \sigma\} \vdash N : \tau}{\Gamma_1 \cup \Gamma_2 \vdash (let\, x \;=\; M\; in\; N) : \tau}\,(let)$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha.\sigma}\,(gen) \qquad \text{Se } \sigma > \sigma' \; \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \sigma'}\,(spec)$$

A especialização ocorre com a regra (*spec*), que se necessário acontece um passo antes da aplicação (*app*). Por exemplo, um termo de tipo $\forall \alpha.\alpha \to \alpha$ aplicado a um termo de tipo τ requer a especialização $\tau \to \tau$ do primeiro termo. Os exemplos abaixo ilustram melhor a necessidade da instanciação.

Exemplo 3.1. Provas em DM

$$\text{(a)} \quad \frac{\dfrac{x : a \vdash x : a}{\vdash (\lambda x.x) : a \to a}\,(abs)}{\vdash (\lambda x.x) : \forall a.a \to a}\,(gen)$$

$$\text{(b)} \quad \frac{\dfrac{i : \forall a.a \to a \vdash i : \forall a.a \to a}{i : \forall a.a \to a \vdash i : (a \to a) \to (a \to a)}\,(spec) \qquad \dfrac{i : \forall a.a \to a \vdash i : \forall a.a \to a}{i : \forall a.a \to a \vdash i : a \to a}\,(spec)}{i : \forall a.a \to a \vdash ii : a \to a}\,(app)$$

$$\text{(c)} \quad \frac{\dfrac{(a)}{\vdash (\lambda x.x) : \forall a.a \to a} \qquad \dfrac{(b)}{i : \forall a.a \to a \vdash ii : a \to a}}{\vdash let\; i \;=\; \lambda x.x\; in\; ii : a \to a}\,(let)$$

$$\text{(d)} \quad \frac{\dfrac{i : \forall a.a \to a \vdash i : \forall a.a \to a}{i : \forall a.a \to a \vdash i : (b \to a \to a) \to (b \to a \to a)}\,(spec) \qquad \dfrac{\dfrac{x : a \vdash x : a}{\vdash \lambda x.x : a \to a}\,(abs)}{\vdash \lambda yx.x : b \to a \to a}\,(abs)}{i : \forall a.a \to a \vdash i(\lambda yx.x) : b \to a \to a}\,(app)$$

$$\text{(e)} \quad \frac{\dfrac{i : \forall a.a \to a \vdash i : \forall a.a \to a}{i : \forall a.a \to a \vdash i : (b \to a \to b) \to (a \to b \to a)}\,(spec) \qquad \dfrac{\dfrac{y : b \vdash y : b}{y : b \vdash \lambda x.y : a \to b}\,(abs)}{\vdash \lambda yx.y : b \to a \to b}\,(abs)}{i : \forall a.a \to a \vdash i(\lambda yx.y) : b \to a \to b}\,(app)$$

$$\text{(f)}\quad \dfrac{\dfrac{(e)}{i : \forall a.a \to a \vdash i(\lambda yx.y) : b \to a \to b} \qquad \dfrac{(d)}{i : \forall a.a \to a \vdash i(\lambda yx.x) : b \to a \to a}}{i : \forall a.a \to a \vdash \langle i(\lambda yx.y), i(\lambda yx.x)\rangle : (b \to a \to b) \times (b \to a \to a)}\,(pair)$$

$$\text{(g)}\quad \dfrac{\dfrac{(a)}{\vdash (\lambda x.x) : \forall a.a \to a} \qquad \dfrac{(f)}{i : \forall a.a \to a \vdash \langle i(\lambda yx.y), i(\lambda yx.x)\rangle : (b \to a \to b) \times (b \to a \to a)}}{\vdash let\ i\ =\ \lambda x.x\ in\ \langle i(\lambda yx.y), i(\lambda yx.x)\rangle : (b \to a \to b) \times (b \to a \to a)}\,(let)$$

A dedução 3.1a, do termo da identidade, é diferente da dedução 2.3a (em TA) pelo passo adicional (opcional) de generalização (*gen*). A segunda dedução (3.1b) revela como duas suposições de termos com operador $\forall$ podem ser instanciadas de maneira distinta, porém a suposição inicial é mantida em ambas e assim o termo final (ii) é dependente do contexto $i : \forall\sigma.\sigma \to \sigma$. A dedução 3.1c mostra como usar regra do operador *let* para liberar a suposição final da dedução 3.1b.

As deduções 3.1d e 3.1e têm conclusões cujos contextos têm o mesmo elemento $i : \forall\sigma.\sigma \to \sigma$, assim em 3.1f a união de ambos os contextos é consistente e o produto de ambos os termos pode ser formado. Por fim, em 3.1g elimina $i : \forall\sigma.\sigma \to \sigma$ do contexto. Isso mostra como i pode ser aplicado a dois tipos diferentes por meio do operador *let*, se for instanciado devidamente para cada. Essa capacidade é conhecida como *let*-polimorfismo.

Em TA é inviável fazer algo equivalente ao *let*-polimorfismo com a equivalência ingênua

$$let\ x\ =\ M\ in\ N \equiv (\lambda x.N)M,$$

pois, apesar de $(\lambda i.(i(\lambda yx.y), i(\lambda yx.x)))(\lambda x.x)$ ter a mesma forma normal que o termo de 3.1g, não é possível tipá-lo em nenhum dos sistemas. O motivo é a inconsistência na união dos contextos de cada um dos elementos do par. O primeiro par do elemento tem o contexto $i : (\alpha \to \sigma \to \alpha) \to (\alpha \to \sigma \to \alpha)$ e o segundo $i : (\alpha \to \sigma \to \sigma) \to (\alpha \to \sigma \to \sigma)$ devido as suposições do termo i.

Um termo pode receber mais de um tipo, desde que o tipo seja condizente com a relação de especialização $>$. Portanto, o tipo $\alpha \to \alpha$ para a função identidade no sistema TA é uma instância de $\forall\alpha.\alpha \to \alpha$ obtida após o uso de (*spec*). Não apenas a identidade, mas qualquer outro termo tipável em TA também é tipável em DM.

Teorema 3.1. Toda prova em TA é uma prova em DM.

Prova:

(i) Os termos de TA são uma sublinguagem dos termos de DM, pois a Definição 2.1 está contida na Definição 3.1.

(ii) Tipos Paramétricos são uma sublinguagem dos Politipos, pois a Definição 2.18 está contida na Definição 3.2.

(iii) As regras de dedução de TA são um subconjunto de DM, pois a Definição 2.22 está contida na Definição 3.5.

Por conseguinte, qualquer prova em TA pode ser reescrita em DM sem alterações. A relação é direta.

Diferentemente de TA, Damas-Milner pretende ser um sistema de tipos para uma linguagem de programação, portanto precisa ser Turing-Completo e apresentar recursão. Porém, por questões de simplicidade o operador de ponto fixo, *fix*, é omitido na linguagem. O operador *fix* deve ter o tipo

$$fix\ : \forall\sigma.(\sigma \to \sigma) \to \sigma.$$

A omissão do operador de ponto fixo (se não adicionado *fix*) é uma evidência de que DM é Fortemente Normalizável, assim como TA. A prova dessa hipótese não será apresentada aqui, porém, para isto bastaria provar que o Sistema F (apresentado a seguir) é Fortemente Normalizável e de que Damas-Milner é, na realidade, um subsistema do Sistema F.

O sistema de regras não fornece um método explícito para inferir o Tipo Principal de um termo, pelo contrário, as regras possibilitam que um conjunto de tipos sejam deduzidos para um mesmo termo. As regras dizem o que é permitido deduzir e mantêm as possibilidades de provas de maneira não-determinística. Em contrapartida, o trabalho de um algoritmo de inferência é fornecer o Tipo Principal.

O Algoritmo W apresentado na Figura 3.1 tem como entrada um contexto (inicialmente vazio) e uma expressão e (termo λ), e retorna o tipo principal de e e uma substituição. O algoritmo funciona de maneira *bottom-up*, e aplica substituições e unificações (*unify*) conforme sobe pela Árvore Abstrata Sintática (*Abstract Syntax Tree* ou AST). Em certos casos, substituições são aplicadas em contextos, o que é intuitivamente feito pela aplicação dessa a cada elemento do contexto. O algoritmo reporta uma falha caso não seja possível inferir o tipo, como em casos de tipos não unificáveis e contextos

inconsistentes. No algoritmo, *fresh* β representa que β é uma variável livre ainda não utilizada e $close(\Gamma, \tau)$ quantifica variáveis α que ocorrem em τ, mas não em Γ. Logo, $\forall \alpha_1 ... \alpha_n . \tau$ caso $\alpha_1 ... \alpha_n \notin \Gamma$.

$W(\Gamma, \texttt{x})$ =

  Se $\Gamma(\texttt{x}) = \forall \alpha_1 ... \alpha_2 . \tau$ então $([\beta_i / \alpha_i]\tau, \texttt{Id})$

  senao Falha

$W(\Gamma, \texttt{e e'})$ =

  let $(\tau, \mathbb{S}_1) = W(\Gamma, \texttt{e})$

    $(\tau', \mathbb{S}_2) = W(\mathbb{S}_1 \Gamma, \texttt{e'})$

    $\mathbb{S}$ = unify $(\mathbb{S}_2 \tau, \tau' \rightarrow \beta)$ onde β fresh

  in $(\mathbb{S}\beta, \mathbb{S} \circ \mathbb{S}_2 \circ \mathbb{S}_1)$

$W(\Gamma, \lambda \texttt{x.e})$ =

  let $(\tau, \mathbb{S}) = W((\Gamma, \{\texttt{x} : \beta\}),)$

  in $(\mathbb{S}(\beta \rightarrow \tau), \mathbb{S})$

$W(\Gamma, \texttt{let x = e in e'})$ =

  let $(\tau, \mathbb{S}_1) = W(\Gamma, \texttt{e})$

    $(\tau\texttt{'}, \mathbb{S}_2) = W(\mathbb{S}_1(\Gamma, \{\texttt{x} : \texttt{close}(\mathbb{S}_1 \Gamma, \tau)\}), \texttt{e'})$

  in$(\tau\texttt{'}, \mathbb{S}_1 \circ \mathbb{S}_2)$

Figura 3.1: Algoritmo W: representação sob a forma de função.

3.1.1 Sistema de Tipos de Haskell

Como qualquer linguagem funcional, Haskell é baseada no Cálculo Lambda. Porém, por razões práticas a sua sintaxe é consideravelmente enriquecida para considerar constantes, relações, operações, funções nominais, declarações de tipos de dados algébricos, classes de tipo etc. Por essa razão, esta subseção será focada nos tipos de Haskell, mais especificamente, nas classes de tipo, e como isso se relaciona com o Sistema Damas-Milner. Uma completa definição da sintaxe (gramática formal) de Haskell é dada em (JONES, 2003).

Algumas diferenças na sintaxe (tipos e termos) em relação ao Damas-Milner são importantes para os exemplos desta seção.

(i) O simbolo $\rightarrow$ é trocado por `->`.

(ii) O simbolo :, que associa um termo com um tipo, é trocado por `::`. $a \rightarrow a$ é representado por `a -> a`.

(iii) Na abstração lambda, o simbolo λ é trocado por \ e o ponto "." por `->`. Portanto, $\lambda x.x$ passa a ser `\x -> x`.

(iv) O operador $\forall$ é omitido em todas as assinaturas de tipo, pois variáveis de tipo sempre estão ligadas no tipo mais geral que é dado pelo algoritmo W. No exemplo abaixo `bar` tem o tipo `a -> b -> a` ($\forall ab.a \rightarrow b \rightarrow a$), porém `foo` tem uma variável livre `x` cujo tipo depende do contexto ($x : a \vdash foo\ : \forall b.b \rightarrow a$).

```
let bar = \ x -> let foo \ y -> x in foo in bar :: a -> b -> a
```

O aspecto computacional que diferencia Haskell de todos os outros modelos de Cálculo Lambda Tipado desta monografia é a possibilidade de escrever programas recursivos. Consequentemente, programas em Haskell podem ser funções parciais, pois existem programas cujas computações não param (não têm forma normal).

Programas recursivos são possíveis graças à adição do combinador de ponto fixo *fix* que satisfaz a propriedade

$$fix\ f\ \equiv_\beta f\ (fix\ f).$$

Haskell redefine termo *let* para

```
let x = M in N ≡ (\x -> N)(fix (\x -> M)),
```

que é conhecido como *let rec* em outras linguagens funcionais como OCaml. Repare que a ordem de avaliação importa para alcançar a forma normal.

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, ou polimorfismo *ad-hoc*, ou polimorfismo com restrições, é necessária quando uma função (seu nome) tem diferentes implementações para diversificados argumentos e quando é necessário definir uma operação somente sobre um grupo específico de tipos.

Dois cenários comuns para a sobrecarrega de funções são as operações aritméticas e a igualdade. As operações aritméticas precisam ter uma implementação para cada tipo de dado numérico: $Int, Float...$ e não podem ser definidas para os demais tipos da linguagem. Equalidade somente pode ser definida para tipos que não são funções, devido à indecibilidade do caso geral da β-equivalência.

A linguagem ML trata sobrecarga de operações aritméticas proibindo que funções definidas em termos dessas sejam usadas de maneira polimórfica, portanto em ML o programa

```
square x = x*x
square 3.13
square 2
```

não é válido. Já Miranda resolve a sobrecarga da aritmética apenas fornecendo o tipo *num* para dados numéricos, que silenciosamente converte para $Float$ ou inteiro de precisão arbitrária de acordo com a necessidade.

Obviamente, ambas abordagens causam restrições inconvenientes. Assim, um dos objetivos do comitê de desenvolvimento da linguagem Haskell foi encontrar um modelo de polimorfismo *ad-hoc* que não colocasse limitações como essas (WADLER; BLOTT, 1989). A solução encontrada foi a adoção das classes de tipo, as quais funcionam como restrições para tipos. Destarte, uma função como a *square* tem tipo

```
square :: Num a => a -> a,
```

onde **Num** é uma classe de tipo que representa todos os tipos numéricos e **Num a** restringe **a** como um tipo que é instância dessa classe. A sintaxe de classes de tipo é dada através do exemplo da Figura 3.2, que deve ser interpretada como "um tipo **a** pertence a classe **Num** se tem funções chamadas de (+), (*) e *negate* com os tipos especificados". As instâncias devem ser lidas como "existem funções (+), (*) e *negate* definidas para o tipo int/float". É tarefa do algoritmo de inferência verificar se os tipos das funções das instâncias estão de acordo com os requisitados pela classe.

```
class Num a where
    (+), (*) :: a -> a -> a
    negate :: a -> a
instance Num Int where
    (+)     = addInt
    (*)     = mulInt
    negate  = negInt
instance Num Float where
    (+)     = addFloat
    (*)     = mulFloat
    negate  = negFloat
```

Figura 3.2: Operações aritméticas sobrecarregadas com classes de tipo.

Linguagens orientadas a objetos geralmente suportam sobrecarga de funções e implementam isso através de um ponteiro do objeto para um dicionário de funções sobrecarregadas. Isso significa que, no caso da equalidade, ambos os objetos estão associados ao mesmo dicionário respectivo. Tal redundância poderia ser evitada se dicionários fossem passados independentemente dos objetos. Esse é o conceito básico da implementação interna das classes de tipo. Mais precisamente, isso é graças a toda classe de tipo poder ser convertida em um novo tipo de dado (`data types`). Em termos de assinatura de tipo: uma função com uma restrição de classe pode ser convertida para uma outra função sem essa restrição e com um tipo que representa essa classe. Dessa maneira, tem-se que

`C a => a -> a` $\equiv$ `CD a -> a -> a`,

onde `CD` é o tipo que representa a classe `C`. Na realidade, essa conversão é usada no processo de compilação.

O procedimento de conversão é dado por meio de um exemplo. Considere a classe de tipo para operações aritméticas da Figura 3.2, a classe `Num` é traduzida para um tipo de dado `NumD` com as assinaturas de tipos de `(+), (*), neg` como parâmetros. As funções `add, mul, neg` são funções que consultam um dicionário. Cada instância da classe `Num` é convertida para uma função que retorna o seu respectivo dado do tipo `NumD` que contém as funções das sobrecargas (Ver Figura 3.3).

```
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a)

add (NumDict a m n) = a
mul (NumDict a m n) = m
neg (NumDict a m n) = n

numDInt :: NumD Int
numDInt = NumDict addInt mulInt negInt

numDFloat :: NumD Float
numDFloat = NumDict addFloat mulFloat negFloat
```

Figura 3.3: Classe de tipo das operações aritméticas traduzida para tipo de dado.

Assim, o compilador realiza a tradução

`2.0 * 1.5` $\Rightarrow$ `mul numDFloat 2.0 1.5`

e o dicionário é passado pelas operações aritméticas em tempo de execução apenas uma vez para uma mesma função sobrecarregada.

Classes funcionam como restrições de tipo impondo a existência de uma implementação. Em certas situações, um mesmo tipo `a` de uma função pode estar sobre a restrição de mais de uma classe, por exemplo

```
numberIn :: Num a, Eq a => a -> [a] -> Bool
```

verifica a existência de um número em uma lista de números. Intuitivamente, colocar essas duas restrições é redundante, afinal todo número é comparável. Há sentido em fazer `Num` uma subclasse de `Eq` conforme Figura 3.4.

```
class Eq a => Num a where
    (+), (*) :: a -> a -> a
    negate :: a -> a
```

Figura 3.4: Exemplo de subclasse para operações aritméticas.

O sistema de classes de Haskell permite que uma classe tenha várias subclasses e, também, que uma classe seja subclasse de várias outras, chamadas de superclasses.

A definição informal de classes de tipo apresentada até agora basta para realizar uma implementação. De fato, o processo de conversão de classes para tipos é suficiente para a implementação do compilador de Haskell *GHC*. Por outro lado, a relevante propriedade de tipo principal neste sistema com classes de tipo é uma questão em aberto e é conjecturada por (WADLER; BLOTT, 1989) de maneira positiva, fornecendo num apêndice a definição formal de classes de tipos que é resumidamente apresentada a seguir.

Para esta formalização não será utilizada a sintaxe de Haskell e sim uma sintaxe como a de Damas-Milner, mas estendida para suportar sobrecarga de identificadores (variáveis) e a declaração de novos tipos. Este sistema chama-se AHDM (Ad-Hoc-Damas-Milner). A sintaxe de tipos e termos é definida com base na gramática da Figura 3.5.

Expressões têm `over` e `inst` para representar, respectivamente, a declaração de identificadores sobrecarregados e 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))
```

tem a mesma função do termo π_0 do sistema TA. Na realidade, a regra de produção `data` generaliza pares e viabiliza definir tuplas de qualquer aridade.

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

Identificadores	$\mathtt{x}$
Expressões	$\mathtt{e} ::= \mathtt{x}$
	$\mid\ \mathtt{e}_0\ \mathtt{e}_1$
	$\mid\ \lambda\mathtt{x}.\mathtt{e}$
	$\mid\ \lambda\vartheta(\mathtt{x}_0...\mathtt{x}_n).\mathtt{e}$
	$\mid\ \mathtt{let}\ \mathtt{x} = \mathtt{e}_0\ \mathtt{in}\ \mathtt{e}_1$
	$\mid\ \mathtt{over}\ \mathtt{x} :: \sigma\ \mathtt{in}\ \mathtt{e}$
	$\mid\ \mathtt{inst}\ \mathtt{x} :: \sigma = \mathtt{e}_0\ \mathtt{in}\ \mathtt{e}_1$
	$\mid\ \mathtt{data}\ \chi(\alpha_0...\alpha_n) = \vartheta(x_0...x_n)\ \mathtt{in}\ \mathtt{e}_1$
Variáveis de tipo	α
Construtor de valor	ϑ
Construtor de tipo	χ
Monotipos	$\tau ::= (\tau \rightarrow \tau') \mid \chi(\tau_0...\tau_n) \mid \alpha$
Tipos predicativos	$\rho ::= (\mathtt{x}::\tau).\rho \mid \tau$
Politipos	$\sigma ::= \forall\alpha.\sigma \mid \rho$

Figura 3.5: Sintaxe de termos e tipos em AHDM.

O sistema de dedução original, apresentado em (WADLER; BLOTT, 1989), 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. Neste dissertação é utilizado uma simplificação do sistema original, que opta pela omissão das conversões. Além disso, foi incluído a regra para deduzir tipo de valores constantes.

O contexto relaciona identificadores com tipos de três maneiras diferentes:

(i) $(x ::_o \sigma)$ para identificadores sobrecarregados;

(ii) $(x ::_i \sigma)$ para instâncias declaradas de identificadores sobrecarregados;

(iii) $(x :: \sigma)$ para variáveis ;

Definição 3.6. Substituição de Tipo

Uma substituição de tipo $\mathbb{S}$ em um tipo predicativo ρ é uma sequência,

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

onde τ_i são quaisquer monotipos e a_i são distintas variáveis de tipo de ρ. A aplicação de $\mathbb{S}$, da Definição 3.4, é estendida por

$$\mathbb{S}((x :: \tau).\rho') \equiv (x :: \mathbb{S}(\tau)).\mathbb{S}(\rho').$$

Nota 3.1. $Eq\ \alpha$ 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

(i) o mesmo predicado $(x :: \tau)$ está em ρ', ou

(ii) o predicado pode ser eliminado no contexto A.

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

(i) $(x :: \tau) \in A$, ou

(ii) $(x ::_i \sigma') \in A$ e $\sigma' >_A \tau$.

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.

Definição 3.7. Regras do Sistema AHDM

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

$$\frac{A \cup \{\vartheta(x_0...x_n) :: \chi(\alpha_0...\alpha_n)\} \vdash e :: x}{A \vdash (\mathbf{data}\,\chi(\alpha_0...\alpha_n)\ =\ \vartheta(x_0...x_n)\,\mathbf{in}\,e) :: x}\,(data)$$

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

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

$$\frac{A \vdash e :: \sigma}{A \vdash e :: \forall\alpha.\sigma}\,(gen) \qquad \frac{A \vdash e :: \forall\alpha.\sigma}{A \vdash e :: [\tau/\alpha]\sigma}\,(spec)$$

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

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

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

Nota 3.2. As regras $(pred)$, (rel) e $(inst)$ tem como condição: $x ::_o \sigma \in A$.

O exemplo de prova abaixo mostra como um operador sobrecarregado pode ser utilizado na definição de uma abstração lambda e em seguida ser aplicado a dois termos. Em especial, a dedução (a) mostra como a abstração lambda tem o tipo especializado com o predicado necessário, a dedução (b) libera o predicado e inclui no contexto a suposição de uma instância, as deduções (c) e (d) aplicam a abstração lambda e, por fim, a dedução (e) elimina do contexto a suposição da instância e da sobrecarga. Nota-se que se a abstração tivesse sido aplicada à dois valores de tipo *Float*, então a regra **inst** não seria capaz de eliminar do contexto a suposição $eq :: Eq\ Float$, pois não há declaração de instância este tipo.

$$
\text{(a)}\quad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{eq\ ::_o\ \forall a.Eq\,a,\ eq :: Eq\,a\ \vdash eq :: Eq\,a}\,(var) \qquad \dfrac{}{x :: a \vdash x :: a}\,(var)}{eq\ ::_o\ \forall a.Eq\,a,\ eq :: Eq\,a,\ x :: a \vdash eq\,x :: a \to Bool}\,(app) \qquad \dfrac{}{y :: a \vdash y :: a}\,(var)}{eq\ ::_o\ \forall a.Eq\,a,\ x :: Eq\,a,\ x :: a,\ y :: a \vdash eq\,x\,y :: Bool}\,(app)}{eq\ ::_o\ \forall a.Eq\,a,\ eq :: Eq\,a,\ x :: a \vdash \lambda y.eq\,x\,y :: a \to Bool}\,(abs)}{eq\ ::_o\ \forall a.Eq\,a,\ eq :: Eq\,a \vdash \lambda xy.eq\,x\,y :: a \to a \to Bool}\,(abs)}{eq\ ::_o\ \forall a.Eq\,a \vdash \lambda xy.eq\,x\,y :: (eq :: Eq\,a).a \to a \to Bool}\,(pred)}{eq\ ::_o\ \forall a.Eq\,a \vdash \lambda xy.eq\,x\,y :: \forall a.(eq :: Eq\,a).a \to a \to Bool}\,(gen)}{eq\ ::_o\ \forall a.Eq\,a \vdash \lambda xy.eq\,x\,y :: (eq :: Eq\,Int).Int \to Int \to Bool}\,(spec)
$$

$$
\text{(b)}\quad
\dfrac{\dfrac{\text{(a)}}{eq\ ::_o\ \forall a.Eq\,a \vdash \lambda xy.eq\,x\,y :: (eq :: Eq\,Int).Int \to Int \to Bool}\,(spec) \qquad \dfrac{}{eq\ ::_i\ Eq\,Int \vdash eq\ ::_i\ Eq\,Int}\,(var)}{eq\ ::_o\ \forall a.Eq\,a,\ eq\ ::_i\ Eq\,Int \vdash \lambda xy.eq\,x\,y :: Int \to Int \to Bool}\,(rel)
$$

$$
\text{(c)}\quad
\dfrac{\dfrac{\text{(b)}}{eq\ ::_o\ \forall a.Eq\,a,\ eq\ ::_i\ Eq\,Int \vdash \lambda xy.eq\,x\,y :: Int \to Int \to Bool} \qquad \dfrac{}{\vdash 1\ ::\ Int}\,(const)}{eq\ ::_o\ \forall a.Eq\,a,\ eq\ ::_i\ Eq\,Int \vdash (\lambda xy.eq\,x\,y)\,1 :: Int \to Bool}\,(app)
$$

$$
\text{(d)}\quad
\dfrac{\dfrac{\text{(c)}}{eq\ ::_o\ \forall a.Eq\,a,\ eq\ ::_i\ Eq\,Int \vdash (\lambda xy.eq\,x\,y)\,1 :: Int \to Bool} \qquad \dfrac{}{\vdash 2\ ::\ Int}\,(const)}{eq\ ::_o\ \forall a.Eq\,a,\ eq\ ::_i\ Eq\,Int \vdash ((\lambda xy.eq\,x\,y)\,1)2 :: Bool}\,(app)
$$

$$
\text{(e)}\quad
\dfrac{\dfrac{\dfrac{}{eq\ ::_o\ \forall a.Eq\,a \vdash eqInt :: Eq\,Int}\,(const) \qquad \dfrac{\text{(d)}}{eq\ ::_o\ \forall a.Eq\,a,\ eq\ ::_i\ Eq\,Int \vdash ((\lambda xy.eq\,x\,y)\,1)2 :: Bool}}{eq\ ::_o\ \forall a.Eq\,a \vdash (\mathbf{inst}\ eq\ ::\ Eq\,Int\ =\ eqInt\ \mathbf{in}\ ((\lambda xy.eq\,x\,y)\,1)\,2) :: Bool}\,(inst)}{\vdash (\mathbf{over}\ eq\ ::\ \forall a.Eq\,a\ \mathbf{in}\ \mathbf{inst}\ eq\ ::\ Eq\,Int\ =\ eqInt\ \mathbf{in}\ ((\lambda xy.eq\,x\,y)\,1)\,2) :: Bool}\,(over)
$$

3.2 Sistema F

O Sistema F (SF), também conhecido como Cálculo Lambda Polimórfico de Segunda Ordem, é uma forma mais flexível de sistema de tipos, no sentido de que o uso polimórfico de funções é menos restrito. Assim como o sistema Damas-Milner, o SF foi descoberto duas vezes de maneira independente: a primeira pelo lógico Jean-Yves Girard em 1972 e a segunda por um cientista da computação chamado John C. Reynolds em 1974. Girard

tinha o interesse de estender o Isomorfismo de Curry-Howard e Reynolds almejava aplicar em linguagens de programação (HANKIN, 2004).

Este sistema é um Cálculo Lambda Tipado misto *a la* Church e *a la* Curry, pois tipos fazem parte da definição de termo e as regras de dedução usam o conceito de Fórmula da Definição 2.21. Um Sistema F puramente *a la* Curry foi mostrado indecidível, na inferência do Tipo Principal, por (WELLS, 1999). Esta é a principal diferença prática em relação ao Damas-Milner, do qual a inferência é decidível. Isso implica que uma linguagem de programação baseada no Sistema F precisa ter explícitas assinaturas de tipos nos programas, porque, apesar da inferência não ser possível, a verificação de tipo pode ser realizada.

O polimorfismo mais flexível do SF pode ser utilizado em Haskell, com as devidas anotações de tipos, por meio da extensão *RankNTypes*, que permite o uso do quantificador de tipo ∀ de maneira irrestrita, em qualquer posição na assinatura de tipos. Em Haskell, de modo geral,

`a -> b -> a` ≡ ∀`a.`(∀`b.(a -> b -> a)` ≡ ∀`a.(a -> (`∀`b.(b -> a))`,

apesar do terceiro não estar de acordo com a sintaxe de tipos, todos têm o mesmo significado: receber um dado do tipo `a`, e retornar uma função que receba um dado de tipo `b` qualquer e retorne um dado do tipo `a` fixado anteriormente. No entanto, existem casos onde o operador ∀, ao lado esquerdo de `->`, resulta numa semântica de tipos que não pode ser implementada em Haskell sem a extensão *RankNTypes*. Exemplificando-se,

(∀`a.a -> a) -> (`∀`b. b -> b)`,

não pode ter o primeiro ∀ movido para o início pois a semântica desse tipo diz que o argumento possivelmente pode ser utilizado de maneira polimórfica, ou seja, é uma função que não foi instanciada/especializada ainda. Portanto, em

```
foo f = fst (f id, f 'a').
```

a função `f` é um exemplo de uso polimórfico de uma função passada como argumento. Já o segundo ∀ pode movido para o início e consequentemente omitido. Observe que isso difere do tipo

∀`a.(`∀`b.((a -> a) -> (b -> b))`,

que apenas exprime que o argumento precisa ser uma função da forma `a -> a` para algum tipo `a`. Em suma, no sistema Damas-Milner o escopo do quantificador é todo politipo, já no Sistema F o escopo pode ser restrito a algumas partes do tipo.

Definição 3.8. Politipo

A seguinte gramática define os tipos do Sistema F:

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

sendo que σ é um tipo polimórfico, c é uma constante de tipo e α é uma variável de tipo.

Diferente de Damas-Milner, variáveis de tipo, tipos funcionais e tipos polimórficos aparecem no mesmo conjunto de regras, significando que o quantificador $\forall$ pode aparecer em qualquer posição de um tipo e não apenas no início. Termos λ estão relacionados ao quantificador $\forall$ por meio do simbolo Λ da nova construção chamada abstração universal. Os termos λ são:

Definição 3.9. Termo λ

$$e ::= x \mid ee' \mid \lambda x^{\sigma}.e \mid e\sigma \mid \Lambda\alpha.e$$

As regras de dedução são similares ao sistema TA, com a explícita anotação de tipo na regra de abstração, a adição da regra de Abstração Universal (*uabs*) e Aplicação Universal (*uapp*). Essas duas regras permitem que um termo consuma um tipo para ser especializado. Assim, (*uabs*) e (*uapp*) funcionam, respectivamente, como uma generalização das regras (*gen*) e (*inst*) da Definição 3.5.

Definição 3.10. Regras do Sistema F

Axiomas:

$$\frac{}{x : \tau \vdash x : \tau}\ (var)$$

Regras de dedução:

$$\frac{\Gamma_1 \vdash e : \alpha \to \tau \qquad \Gamma_2 \vdash e' : \alpha}{\Gamma_1 \cup \Gamma_2 \vdash (ee') : \tau}\ (app) \qquad \frac{\Gamma \cup \{x : \sigma\} \vdash e : \tau}{\Gamma \vdash (\lambda x^{\sigma}.e) : \sigma \to \tau}\ (abs)$$

$$\frac{t \notin FTV(\Gamma) \quad \Gamma \vdash e : \sigma}{\Gamma \vdash (\Lambda t.e) : \forall t.\sigma}\,(uabs) \qquad\qquad \frac{\Gamma \vdash e : \forall t.\sigma}{\Gamma \vdash e\tau : [\tau/t]\sigma}\,(uapp)$$

O exemplo 3.2a mostra como o termo identidade pode ser quantificado pela Abstração Universal e depois aplicado a um tipo $\alpha \to \alpha$ pela Aplicação Universal. A autoaplicação da identidade, termo II, do exemplo 3.2b serve como comparação com o polimorfismo do quesito 3.1c, demostrando como o uso de funções quantificadas no SF acontece sem o termo let.

Exemplo 3.2. Provas em SF

$$\text{(a)} \quad \frac{\dfrac{\dfrac{x : a \vdash x : a}{\vdash \lambda x^a.x : a \to a}\,(abs)}{\vdash \Lambda a \lambda x^a.x : \forall a.a \to a}\,(uabs)}{\vdash (\Lambda a \lambda x^a.x)(\alpha \to \alpha) : (\alpha \to \alpha) \to (\alpha \to \alpha)}\,(uapp)$$

$$\text{(b)} \quad \frac{\dfrac{(a)}{\vdash (\Lambda a \lambda x^a.x)(\alpha \to \alpha) : (\alpha \to \alpha) \to (\alpha \to \alpha)} \quad \dfrac{x : \alpha \vdash x : \alpha}{\vdash \lambda x^\alpha.x : \alpha \to \alpha}\,(abs)}{\vdash ((\Lambda a \lambda x^a.x)(\alpha \to \alpha))(\lambda x^a.x) : \alpha \to \alpha}\,(app)$$

4 Teoria das Categorias

A Teoria das Categorias foi criada, em 1945, por Samuel Eilenberg e Sanders Mac Lane (EILENBERG; LANE, 1945) como uma maneira de compreender o processo de preservação das estruturas matemáticas na Topologia Algébrica. Segundo (BARR; WELLS, 1995), Categorias podem ser entendidas como uma maneira de descrever a passagem de uma estrutura matemática para outra, como a formalização da descrição de um tipo de estruturas matemáticas (álgebras, lógicas, etc). Todas essas visões são úteis para diversas áreas da Matemática e a última é a mais relevante para a proposta desta monografia, pois lida com a semântica e as propriedades de tipos de estruturas.

Categorias tratam com objetos, morfismos e a composição de morfismos. Objetos são representados por pontos ou letras e morfismos são representados por setas $\rightarrow$, as quais representam qualquer tipo de transformação entre os objetos. Há diversas vantagens na abordagem categórica de teorias matemáticas (conjuntos, grupos, lógicas, tipos, etc), como dualidade, herança de resultados, notação gráfica e a expressividade das construções as quais permitem expressar estruturas complexas de forma simples (MENEZES; HAEUSLER, 2001).

A finalidade da Teoria das Categorias, neste ensaio, é servir como uma fundação matemática de uma pura teoria de funções para explicar sistemas de tipos. Como dito por (SCOTT, 1980), "*O que estamos provavelmente procurando é uma 'mais pura' visão de funções: uma teoria de funções em si mesmas, não uma teoria de funções derivada de conjuntos. O que, então, é uma pura teoria de funções? Resposta: Teoria das Categorias.*" Pode-se pensar ainda que uma pura teoria de funções é uma visão limitada, afinal as aplicações de categorias vão além de explicar funções.

Objetiva-se identificar tipos de estruturas e estruturas categóricas em sistemas de tipos, especificamente no sistema da linguagem Haskell. A relação entre Haskell e Teoria das Categorias deste trabalho é similar ao feito com ML por (RYDEHEARD; BURSTALL, 1988), em que conceitos de categorias são representados como tipos e provas construtivas em Teoria das Categorias viram programas. Diferente de ML, Haskell tem alguns elementos categóricos realizados por classes de tipos, como as mônadas, introduzidas por

Philip Wadler e Simon Peyton-Jones, em Maio de 1996, como uma maneira de evitar efeitos colaterais em I/O (HUDAK et al., 2007). Na realidade, Eugenio Moggi foi o primeiro a apresentar uso de mônadas para representar efeitos em computações em semântica denotacional (MOGGI, 1991).

A relação de Teoria das Categorias com Sistemas de Tipos ficará mais clara com a apresentação da semântica categórica do Cálculo Lambda Tipado, no próximo capítulo, descrita por (BELL, 2012) como uma teoria de tipos sem sua sintaxe, ou seja, somente o significado.

4.1 Categorias

Categorias são formadas por objetos (pontos), morfismos (setas) e uma forma de compor morfismos de maneira associativa. Objetos não necessariamente são conjuntos ou elementos de um conjunto, na realidade não existe restrição do que eles podem representar. Cada objeto deve ao menos ter um morfismo identidade, cujo origem e destino são o próprio objeto em questão.

Definição 4.1. Categoria

Uma categoria $\mathbf{C}$ é definida pela quintupla: $\langle Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \partial_0, \partial_1, \iota, \circ \rangle$ onde,

(a) $Obj_{\mathbf{C}}$ é a coleção de objetos da categoria $\mathbf{C}$, denotado por $A, B, C...$

(b) $Mor_{\mathbf{C}}$ é a coleção de morfismos da categoria $\mathbf{C}$, denotados por $f : A \to B, g : A \to C, h : B \to B...$, onde A, B e C são quaisquer objetos em $Obj_{\mathbf{C}}$.

(c) ∂_0 e ∂_1 são as funções que levam, respectivamente, um morfismo ao seu objeto de domínio e codomínio.

(d) $\circ$ é a função que leva dois morfismos ao seu morfismo de composição resultante, de maneira que respeite a associatividade, ou seja, dados três morfismos $h : A \to B, g : B \to C$ e $f : C \to D$

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

(e) ι é a função que retorna o morfismo identidade, representado por Id_A, de um dado objeto A. O morfismo identidade deve ser tal que para dados dois objetos A e B,

seus respectivos morfismos identidade Id_A e Id_B e um morfismo $f : A \to B$, então a *propriedade da identidade*

$$f \circ Id_A = Id_B \circ f = f$$

é verdadeira.

Exemplo 4.1. Categoria **Set**

O primeiro exemplo é a categoria **Set**, onde objetos são conjuntos e morfismos são funções totais. Note que não foi pormenorizado algum conjunto, então $Obj_{\mathbf{Set}}$ é a coleção de todos os conjuntos, que não pode ser um conjunto. Uma categoria à qual Obj não é um conjunto é chamada de *Grande* (e *Pequena* caso contrário).

A categoria **Set** é definida por

(a) $Obj_{\mathbf{Set}}$ é a coleção de todos os conjuntos.

(b) $\mathrm{Mor}_{\mathbf{Set}}$ é a coleção de todas as funções totais.

(c) ∂_0 e ∂_1 são as funções que levam, respectivamente, uma função f ao seu conjunto de origem e destino.

(d) $\circ$ é a tradicional operação de composição de funções. A associatividade da composição de funções é provada para $f : A \to B$, $g : B \to C$, $h : C \to D$ e $\forall a \in A$ como segue:

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

(e) ι é função que retorna a função identidade de cada conjunto. A função identidade é provada existir pela simples possibilidade de construir uma função que sempre mapeia cada elemento nele mesmo. A prova da *propriedade da identidade* é feita para duas funções identidades quaisquer Id_A e Id_B, e uma função qualquer $f : A \to B$ onde existe algum mapeamento tal que $f(a) = b$, então

$$f \circ Id_A(a) = f(a) = b = Id_B \circ f(a)$$

Um conjunto parcialmente ordenado também pode ser visto como uma categoria. Por exemplo, $\langle \{0, 1, 2\}, \leq \rangle$ é um conjunto parcialmente ordenado cujos objetos são $\{0, 1, 2\}$ e os morfismos são definidos pela relação $\leq$, ou seja, $\{(0 \leq 0), (0 \leq$

$1), (1 \leq 1), (1 \leq 2), (2 \leq 2), \}$. A composição permite definir os morfismos restantes: $\{(0 \leq 2) = (1 \leq 2) \circ (0 \leq 1)\}$. É importante observar que uma relação de ordem é verdadeira ou falsa para quaisquer dois elementos, assim é capaz apenas de gerar nenhum ou apenas um morfismo para esses dois objetos (elementos). Consequentemente, o morfismo de um objeto para ele mesmo somente pode ser a identidade.

Exemplo 4.2. Categoria **Prog**

Uma categoria pode ser definida para representar os tipos e funções de uma linguagem de programação funcional. Toma-se os tipos primitivos (*nat, char, bool...*) como tipos e funções (*+, -, /, chr, ord...*) como morfismos. A noção de elemento é resgatada ao considerar-se como tipo um conjunto unitário representado por 1. Seja $1 \to A$ o tipo de função que parte de 1 e chega ao tipo A, então a quantidade de funções (mapeamentos) totais possíveis com tal assinatura é o número de elementos que existem no conjunto representado por A. A título de exemplo, o conjunto de valores do tipo *bool* é $\{true, false\}$, portanto os mapeamentos possíveis são como mostra a Figura 4.1. Caso mapeamentos parciais fossem aceitos, então existiriam mais formas de mapear 1 em *bool*.

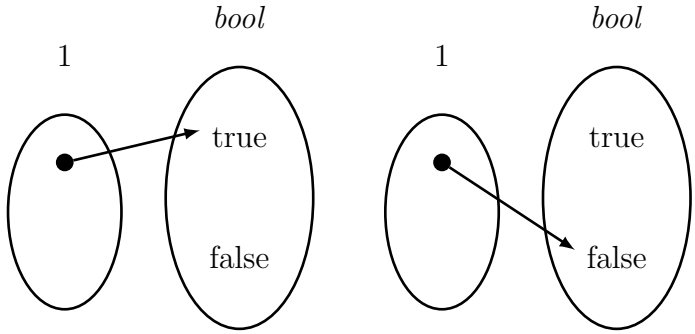

Figura 4.1: Mapeamentos possíveis para as funções de tipo $1 \to$ *bool*.

A categoria de uma linguagem de programação funcional, denotada por **Prog**, é definida por

(a) $Obj_{\mathbf{Prog}}$ é o conjunto de todos os tipos de dados da linguagem: {char, nat, bool, 1, void ...}, onde nat é o tipo que representa os números naturais e *void* representa o conjunto vazio, por exemplo.

(b) $\text{Mor}_{\mathbf{Prog}}$ são todas as funções da linguagem: {+, -, succ, ord, chr, empty ...}, onde succ é a função sucessora, por exemplo. Para que constantes possam ser representadas por morfismos, é necessário que todas as funções sejam totais.

(c) ∂_0 e ∂_1 são as funções que levam, respectivamente, uma função f ao seu tipo de origem e destino.

(d) $\circ$ é o construtor de operações da linguagem que serve como uma maneira de compor as funções. Os tipos das composições devem sempre estar de acordo com a composição pretendida, ou seja, para criar uma composição $g \circ f : A \to C$, é indispensável que $f : A \to B$ e $g : B \to C$.

(e) ι retorna a função identidade de cada tipo função. A função de identidade de cada tipo é a operação vazia, aquela que retorna o próprio valor de entrada. É trivial apurar que para todos os tipos A e B da linguagem, existem funções Id_A e Id_B que retornam o próprio elemento recebido e que para qualquer função $f : A \to B$

$$f \circ Id_A = Id_B \circ f.$$

O objeto *void* é o conjunto vazio e portanto a função vazia, $empty : void \to A$, é única função que parte de *void* e chega em qualquer outro objeto A. Como *void* não tem elemento, não existe como fornecer um argumento para *empty*, logo *empty* é uma função blefe, pois faz-se crer em algo que não pode ser usado.

A representação gráfica parcial da categoria **Prog** está apresentada na Figura 4.2. Nota-se que a categoria não diz nada sobre a sintaxe da linguagem e sim somente sobre os significados que podem ser expressados nela, logo a Teoria das Categorias pode ser utilizada para representar a semântica de uma linguagem de programação.

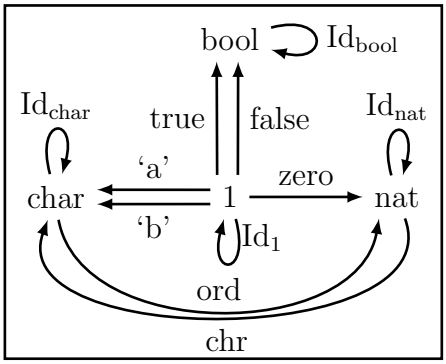

Figura 4.2: Representação parcial da categoria de uma linguagem de programação.

Exemplo 4.3. Categoria **Imp**

Considere o fragmento de lógica proposicional onde há um conjunto de símbolos $A = \{p, q, w, z...\}$ que representam proposições, um conjunto de símbolos $\Omega = \{\vee, \wedge, \to\}$

que são, respectivamente, os conectivos lógicos de disjunção, conjunção e implicação, e a única regra de derivação é o silogismo hipotético:

$$\frac{P_1 \to P_2 \qquad P_2 \to P_3}{P_1 \to P_3}$$

onde P_1, P_2 e P_3 são proposições.

Uma categoria pode ser definida de maneira que retrate todos as possíveis formulações desse fragmento de lógica proposicional. Para isso, é necessário considerar as fórmulas como objetos e as implicações como morfismos. A definição exata é como segue:

(a) $Obj_{\mathbf{Imp}}$ é a coleção de todas as fórmulas lógicas $(p, q, p \vee q, p \wedge q, p \to q...)$.

(b) $\mathrm{Mor}_{\mathbf{Imp}}$ é a coleção de todas as implicações entre fórmulas lógicas $(p \to q, (p \vee q) \to w, p \to p...)$.

(c) ∂_0 e ∂_1 são as funções que levam, respectivamente, uma implicação à sua hipótese e tese.

(d) A composição de morfismos é a regra de derivação silogismo hipotético. A prova da associatividade é dada para todas proposições P_1, P_2, P_3 e P_4, onde ambas as derivações abaixo representam a associatividade das implicações $P_1 \to P_2, P_2 \to P_3$ e $P_3 \to P_4$.

$$\frac{\dfrac{P_1 \to P_2 \qquad P_2 \to P_3}{P_1 \to P_3} \qquad P_3 \to P_4}{P_1 \to P_4} \qquad\qquad \frac{P_1 \to P_2 \qquad \dfrac{P_2 \to P_3 \qquad P_3 \to P_4}{P_2 \to P_3}}{P_1 \to P_4}$$

(e) ι é função que retorna a implicação identidade de cada proposição. A prova da existência de uma implicação identidade Id_{P_1} para qualquer proposição P_1 pode ser obtida por meio da construção de uma tabela verdade. A prova da *propriedade da identidade* é feita para quaisquer proposições P_1 e P_2. Assim, tem-se que

$$\frac{P_1 \to P_1 \qquad P_1 \to P_2}{P_1 \to P_2} \qquad\qquad \frac{P_1 \to P_2 \qquad P_2 \to P_2}{P_1 \to P_2}$$

4.2 Diagramas

Um *diagrama* D de uma categoria $\mathbf{C}$ é composto por nós (objetos) e setas (morfismos). É representado por um grafo (desenho), que fornece uma maneira conveniente de ilustrar as propriedades da categoria $\mathbf{C}$. Diferente da representação parcial de uma categoria, como na figura 4.2, é possível que num *diagrama*:

(i) objetos e setas se repitam.

(ii) morfismos e objetos sejam ocultados.

Definição 4.2. Diagrama

Um diagrama D, de uma categoria $\mathbf{C}$, é uma dupla $\langle A, M \rangle$, tal que

(i) $A = \{A_i | i \in I_D\}$[1] é uma multicoleção de objetos da categoria $\mathbf{C}$.

(ii) $M = \{d_e | e \in E_D\}$ é uma multicoleção de morfismos da categoria $\mathbf{C}$, sendo que

$$\iota_0(d_e) \in A$$
$$\iota_1(d_e) \in A.$$

A vantagem de caracterizar propriedades por meio de diagramas é a alta expressividade. De outra maneira, seria necessário representar uma propriedade por diversas equações. Por exemplo, considere que numa categoria qualquer $\mathbf{C}$ existem os objetos A, B e C, e os morfismos h, f e g, tais que

$$h = g \circ f$$

Então, é possível representar essa equação pelo diagrama da Figura 4.3.

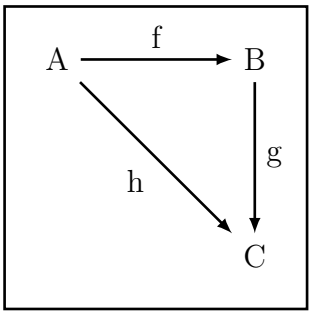

Figura 4.3: Diagrama Comutativo.

[1]Por facilidade utiliza-se a notação de conjuntos para coleções/multicoleções.

Definição 4.3. Diagrama Comutativo

Seja **C** uma categoria qualquer, a qual $A, B \in Obj_{\mathbf{C}}$, então um diagrama comutativo d, com os mesmos objetos A e B, é um desenho o qual quaisquer caminhos alternativos resultantes da composição das setas entre A e B são iguais. Assim, se uma propriedade p pode ser figurada por um diagrama comutativo d, e d é um diagrama da categoria **C**, então **C** tem a propriedade p.

4.3 Subcategorias, Funtores e Categorias de Categorias

Uma categoria existe se, e somente se, está de acordo com a definição 4.1, portanto nada impede uma categoria conter outras categorias. Essa é uma das principais características da Teoria das Categorias: a possibilidade de construir categorias complexas a partir das simples (ASPERTI; LONGO, 1991).

Definição 4.4. Subcategoria

Um subcategoria de uma categoria **C** é qualquer parte de **C** que satisfaça a definição de categoria. Para isso, seja **S** uma subcategoria de **C**, então

(a) $Obj_{\mathbf{S}} \subseteq Obj_{\mathbf{C}}$.

(b) Para todos os objetos A e B de $Obj_{\mathbf{C}}$, $Hom_{\mathbf{S}}(A, B) \subseteq Hom_{\mathbf{C}}$(A, B), onde $Hom_{\mathbf{X}}$(A, B) é a função que retorna uma coleção com todos os morfismos de A para B na categoria **X**.

(c) A composição e a identidade de **S** são as mesmas de **C**.

Dadas duas categorias **A** e **B**, se existe uma maneira de mapear os objetos e os morfismos de **A** em **B**, e além disso, se tal mapeamento preserva origem, destino, identidade e composição, então esse mapeamento é chamado de *funtor covariante* (ou apenas *funtor*) da categoria **A** para **B**.

Definição 4.5. Funtor

Sejam **A** e **B** categorias, então **F** é um funtor de **A** para **B** se, e somente se,

$$\mathbf{F} = \langle F_O, F_M \rangle,$$

onde $F_O : Obj_{\mathbf{A}} \to Obj_{\mathbf{B}}$ e $F_M : Mor_{\mathbf{A}} \to Mor_{\mathbf{B}}$, de maneira que

(i) Se $f_A : X \to Y$ é um morfismo de $\mathbf{A}$, então $F_M(f_A) : F_O(X) \to F_O(Y)$ é um morfismo de $\mathbf{B}$.

(ii) Se Id_X é a identidade de algum objeto X de $\mathbf{A}$, então $F_M(Id_{F_O(X)})$ é a identidade do objeto $F_O(X)$ na categoria $\mathbf{B}$.

(iii) Se $g \circ_A f : X \to Y$ é uma composição de morfismos em $\mathbf{A}$, então $F_M(g) \circ_B F_M(f) : F_O(X) \to F_O(Y)$ é uma composição de morfismos em $\mathbf{B}$.

Uma possibilidade da Teoria das Categorias é trabalhar com vários níveis de abstração, isto é, formar uma categoria a partir de algo (conjuntos, aritmética, grupos...) e formar outras categorias que têm essa como objetos. Uma categoria que tem categorias como objetos e funtores entre as mesmas é chamada de *Categoria de Categorias*.

A Categoria de Categorias que tem como objetos categorias pequenas e como morfismos funtores é conhecida como **Cat**.

4.4 Categoria Dual e Dualidade

O conceito de dualidade também permite construções de novas categorias a partir de outras já existentes. A simples inversão de sentido das flechas dos morfismos de uma categoria $\mathbf{C}$ cria uma nova categoria chamada *categoria dual* de $\mathbf{C}$, denotada por $\mathbf{C}^{op}$. As propriedades duais da categoria original são herdadas na sua dual, isso significa que o trabalho realizado numa categoria é ganho em sua dual. Assim, seja um conceito expresso pela proposição P e o seu conceito dual expresso pela proposição P^{op}, então são verdadeiras construções da forma: "Se P é verdadeiro na categoria $\mathbf{C}$, então P^{op} é verdadeiro em $\mathbf{C}^{op}$".

Definição 4.6. Categoria Dual

Uma categoria $\mathbf{C}^{op}$ é a categoria dual de $\mathbf{C}$ se, e somente se,

(i) os objetos e morfismos são os mesmos.

(ii) as funções ι_0 e ι_1 em $\mathbf{C}$ são, respectivamente, ι_1 e ι_0 em $\mathbf{C}^{op}$.

(iii) a composição $g \circ f = h$ em $\mathbf{C}$ é a composição $f \circ g = h$ em $\mathbf{C}^{op}$.

4.5 Objeto Inicial e Objeto Terminal

A categoria de linguagens funcionais **Prog** tem o objeto 1, que a priori parece não ser necessário. Não obstante, como demonstrado, 1 permite resgatar o conceito de elemento na Teoria das Categorias. Outra peculiaridade do objeto 1 em **Prog** é que a quantidade de morfismos que partem de um outro objeto qualquer e chegam nele é sempre um. Um objeto em que sempre é apontado por um único morfismo a partir de qualquer outro objeto é chamado de objeto terminal, portanto 1 é objeto terminal de **Prog**. Na categoria **Set**, diferente de **Prog**, o objeto terminal não é único, são todos os conjuntos unitários.

O dual do objeto terminal é o objeto inicial, do qual parte um único morfismo para qualquer outro objeto. Na categoria **Set**, o objeto inicial é o conjunto vazio, pois só existe um morfismo possível que parte dele e chega a qualquer outro objeto: a função vazia.

Definição 4.7. Objeto Inicial

Para qualquer categoria $\mathbf{C}$, onde $0 \in Obj_{\mathbf{C}}$, e para objeto qualquer $A \in \mathbf{C}$, então 0 é o objeto inicial se, e somente se, existe um único morfismo

$$h : 0 \to A.$$

Definição 4.8. Objeto Terminal

Para qualquer categoria $\mathbf{C}$, onde $1 \in Obj_{\mathbf{C}}$, e para objeto qualquer $A \in \mathbf{C}$, então 1 é o objeto terminal se, e somente se, existe um único morfismo

$$h : A \to 1.$$

O objeto terminal pode ser usado para "resgatar" a noção de elemento, como visto em **Prog**, a partir de morfismos que partem do objeto terminal. Por outro lado, caso o objeto terminal seja o mesmo do inicial, então o resgate pode não acontecer. Por exemplo, na categoria **Pfn**, que é igual a **Set**, mas com funções parciais, o objeto terminal e inicial são o conjunto vazio e a quantidade de funções que partem do conjunto vazio é sempre um (a função vazia).

4.6 Produto e Coproduto

Produto na Teoria das Categorias é uma maneira de generalizar o conceito de produto cartesiano da Teoria dos Conjuntos, tal que

$$A \times B = \{(a, b) \mid a \in A, b \in B\}.$$

Isso permite a representação de operações de aridades arbitrárias dentro da Teoria das Categorias [1]. A existência de tal estrutura, que será definida a seguir, dentro de uma categoria significa que é possível definir produto cartesiano entre objetos, mesmo que a categoria em questão não seja algum tipo de Teoria dos Conjuntos. Essa é outra característica da Teoria das Categorias, a possibilidade de tratar as propriedades independentemente da implementação (MENEZES; HAEUSLER, 2001) ou conforme (ASPERTI; LONGO, 1991) "a descrição categórica corresponde a uma especificação de um dado abstrato"[2].

Definição 4.9. Produto

Seja uma categoria $\mathbf{C}$ e objetos A, $B \in Obj_{\mathbf{C}}$, um Produto entre A e B $(A \times B)$ na categoria $\mathbf{C}$ é composto pelo objeto $A \times B$, e pelos morfismos $\pi_1 : A \times B \to A$ e $\pi_2 : A \times B \to B$, se, e somente se, para qualquer objeto $C \in Obj_{\mathbf{C}}$, e todo par de morfismos $f : C \to A$ e $g : C \to B$, existe um único morfismo $h : C \to A \times B$ tal que o diagrama representado na Figura 4.4 comute.

.

$$\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[d, dashed, "h"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B
\end{tikzcd}$$

Figura 4.4: Diagrama comutativo do produto entre A e B.

O dual do Produto é o Coproduto, que permite a generalização da soma dis-

[1] Aridade arbitrária também foi construída no Cálculo Lambda por meio de Currying.

[2] Uma das metas deste trabalho é tentar identificar estruturas categóricas em tipos de dados na linguagem Haskell.

junta (ou união disjunta) entre dois conjuntos. A soma disjunta é dada por

$$A + B = \{(a, 0) \mid a \in A\} \cup \{(b, 1) \mid b \in B\}.$$

Definição 4.10. Coproduto

Seja uma categoria $\mathbf{C}$ e objetos A, $B \in Obj_{\mathbf{C}}$, um Coproduto entre A e B $(A+B)$ na categoria $\mathbf{C}$ é composto pelo objeto $A+B$, e pelos morfismos $\pi_1 : A \to A+B$ e $\pi_2 : B \to A + B$, se, e somente se, para qualquer objeto $C \in Obj_{\mathbf{C}}$, e todo par de morfismos $f : A \to C$ e $g : B \to A$, existe um único morfismo $h : A + B \to C$, tal que o diagrama representado na Figura 4.5 comute.

$$\begin{tikzcd}
A \arrow[r, "\pi_1"] \arrow[dr, "f"'] & A + B \arrow[d, dashed, "h"] & B \arrow[l, "\pi_2"'] \arrow[dl, "g"] \\
& \mathrm{C} &
\end{tikzcd}$$

Figura 4.5: Diagrama Comutativo do coproduto entre A e B.

4.7 Cones e Cocones

Cones e Cocones são estruturas categóricas que podem acontecer para um dado diagrama D de uma categoria $\mathbf{C}$.

Definição 4.11. Cone

Sobre um diagrama $D = \langle A, M \rangle$, de uma categoria $\mathbf{C}$, um cone $\langle K, F \rangle$ é um objeto $K \in Obj_{\mathbf{C}}$ (vértice) e uma multicoleção de morfismos (pernas) $F = \{f_i \mid i \in I_D\}$ de $\mathbf{C}$, com $f_i : K \to A_i$ para todo $i \in I_D$, tal que

$$\forall i, j \in I_D.\, \forall e \in E_D.\, d_e : A_i \to A_j \in M \Longrightarrow d_e \circ f_i = f_j.$$

Tal construção também pode ser representada por um diagrama comutativo como mostra a Figura 4.6. Cada sequência de setas comutativas formam um triângulo com os objetos no diagrama, e estes juntos formam um desenho que lembra um cone, cujo vértice é o objeto K.

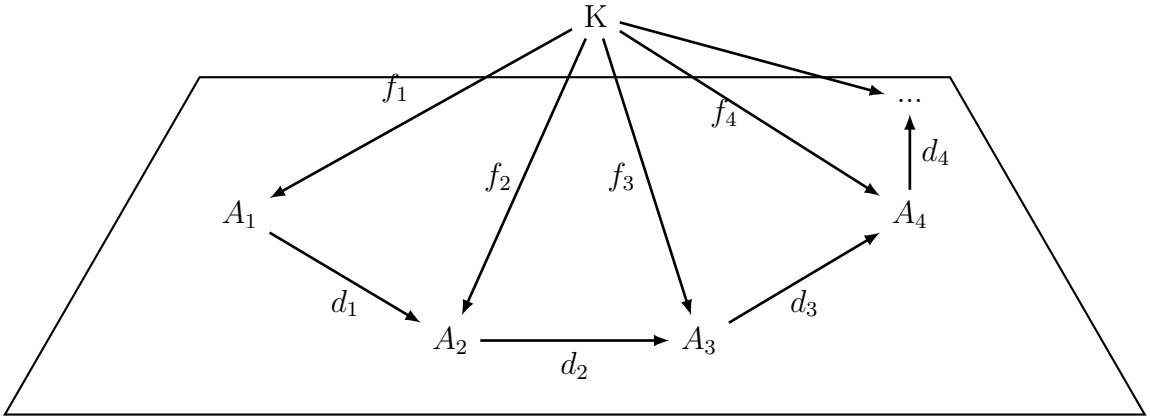

Figura 4.6: Representação de um *Cone* sobre um diagrama D.

As seguintes situações estão de acordo com a definição de cone:

(a) $D = \langle \emptyset, \emptyset \rangle$ e um cone $\langle K, \emptyset \rangle$ (ver desenho da esquerda da Figura 4.7), onde K é qualquer objeto da categoria do diagrama.

(b) $D = \langle \{A, B\}, \{f_2, f_3\} \rangle$ e um cone $\langle K, \{f_1, d\} \rangle$ (ver desenho do meio da Figura 4.7) , onde o morfismo d é tal que $d = f_2 \circ f_1 = f_3 \circ f_1$.

(c) $D = \langle \{A, B\}, \emptyset \rangle$ e um cone $\langle K, \{f_1, f_2\} \rangle$ (ver desenho da direita da Figura 4.7).

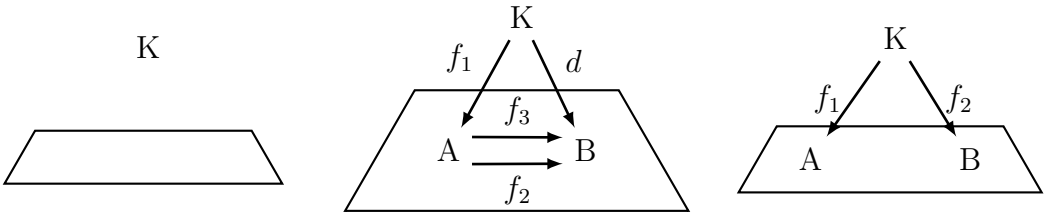

Figura 4.7: Diagrama sem objetos e morfismos (esquerda), diagrama com dois objetos e dois morfismos paralelos (meio), e diagrama com dois objetos e sem morfismos (direita).

O dual de Cone é Cocone, que é dado pela reversão das setas de um cone, ou de maneira mais formal pela seguinte definição:

Definição 4.12. Cocone

Sobre um diagrama $D = \langle A, M \rangle$, de uma categoria $\mathbf{C}$, um cocone $\langle K, F \rangle$ é um objeto $K \in Obj_{\mathbf{C}}$ (vértice) e uma multicoleção de morfismos (pernas) $F = \{f_i \mid i \in I_D\}$ de $\mathbf{C}$, com $f_i : K \to A_i$ para todo $i \in I_D$, tal que

$$\forall i, j \in I_D. \, \forall e \in E_D. \, d_e : A_i \to A_j \in M \Longrightarrow f_j \circ d_e = f_i.$$

4.8 Limites e Colimites

Limites e Colimites são úteis para explicar as ideias de *construção universal* e *propriedade universal*. São, respectivamente, a ideia de que certas entidades são definidas com base na existência e unicidade de uma seta; e que existe uma construção ótima que satisfaz uma certa propriedade, a qual tem todas as informações necessárias para definir qualquer outra construção de mesmo tipo de forma única.

Na realidade, alguns exemplos de construções universais já foram dados: objeto inicial/terminal e produto/coproduto. É notável que essas construções aparecem aos pares, sempre com o seu dual. Assim, a definição de limite e colimite agrega mais uma construção universal.

Se uma construção de um certo tipo satisfaz uma propriedade universal, então tem-se uma entidade que melhor representa todas as construções desse tipo, uma *construção universal*. Um exemplo são os divisores em comum de um par de números: seja o par 36 e 90, seus divisores em comum são: $1, 2, 3, 6, 9$ e 18. Qualquer um desses números satisfaz a propriedade de ser divisor comum entre 36 e 90, porém o 18 é o número que melhor representa todos os divisores, uma vez que é divisível por todos outros divisores. Considere o desenho da Figura 4.8, onde uma seta $A \to B$ significa B é divisível por A.

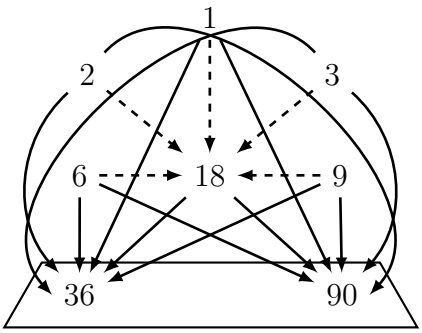

Figura 4.8: Maior divisor comum entre 36 e 90.

Esse exemplo serve como uma noção intuitiva do que é um limite. Todos os divisores comuns são como o objeto que representa um vértice de um cone sobre um diagrama com os objetos 36 e 90, e em especial o 18 é o vértice cujo cone é um limite. A noção de limite é a de um melhor cone de um diagrama.

Definição 4.13. Limite

Suponha um diagrama D de uma categoria $\mathbf{C}$ e o conjunto Cos de todos os seus cones. Se em Cos existe um cone $L = \langle P, \{p_i : P \to A_i \,|\, i \in I_D\}\rangle$ de maneira que para qualquer outro cone $\langle K, \{k_i : K \to A_i \mid i \in I_D\ \rangle\}$ existe um único morfismo $h : K \to P$ tal que

$$\forall i \in I_D.\, p_i \circ h = k_i,$$

então L é o limite sobre diagrama D.

Uma definição alternativa é dada por meio da categoria $\mathbf{Cones}_D$ definida sobre um diagrama D de uma categoria $\mathbf{C}$, onde objetos são cones $\langle K, \{k_i : K \to A_i \mid i \in I_D\ \}\rangle$ e morfismos são os morfismos $h : K \to P$ de $\mathbf{C}$, tais que

$$\forall i \in I_D.\, p_i \circ h = k_i,$$

Assim o limite do diagrama D pode ser visto como o objeto terminal de $\mathbf{Cones}_D$.

Um exemplo de limite é o produto num diagrama com dois objetos e nenhum morfismo entre eles, como o lado direito da Figura 4.7 mostra. Se em tal diagrama existe um cone que satisfaz a propriedade do limite, então esse limite é um produto. Ou seja, existe um vértice P em que sempre chega um único morfismo h a partir de outro vértice K qualquer (ver Figura 4.9).

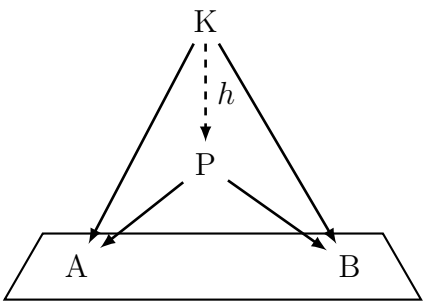

Figura 4.9: Produto como limite do diagrama com dois objetos.

Como toda construção universal tem o seu dual, o dual do Limite é o Colimite.

Definição 4.14. Colimite

O colimite é o objeto inicial da categoria $\mathbf{Cones}_D$ para um diagrama D de uma categoria $\mathbf{C}$.

4.9 Equalizador e Coequalizador

Equalizador e Coequalizador são, respectivamente, limites e colimites especiais que existem em diagramas com morfismos paralelos como no desenho do meio da Figura 4.7.

A explicação intuitiva pode ser dada mediante a categoria **Set**, onde só há funções totais. Um equalizador, em **Set**, é o maior conjunto A_0 que "equaliza ou iguala" duas funções $f, g : A \to B$ como ilustra a Figura 4.10. A função $inc : A_0 \to A$ é conhecido como inclusão, pois caracteriza $A_0 \subseteq A$. Do ponto de vista categórico, A_0 é o vértice do cone limite, inc é uma perna. Os conjuntos A e B, e as funções f, g são, na devida ordem, objetos e morfismos em diagrama D.

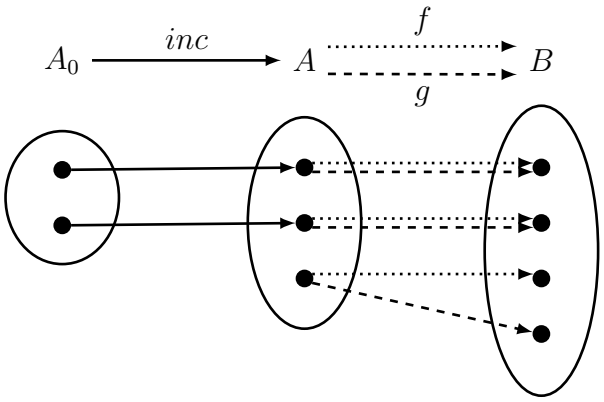

Figura 4.10: Conjunto A_0 equalizador das funções f e g.

Definição 4.15. Equalizador

Em um diagrama D de uma categoria $\mathbf{C}$, composto por um par de objetos A, B e um par de morfismos paralelos $f, g : A \to B$, um equalizador de f, g é um cone $\langle P, \{p : P \to A\}\rangle$ tal que, para qualquer cone $\langle K, \{k : K \to A\}\rangle$ existe um único morfismo $h : K \to P$, no qual

$$p \circ h = k.$$

Na categoria **Prog** o par de morfismos paralelos (true : 1 → bool, false : 1 → bool) tem o equalizador $\langle void, \{empty : void \to bool\}$, pois a intersecção das imagens de true : 1 → bool e false: 1 → bool é vazia, portanto o maior conjunto que é capaz de equalizá-los é o conjunto vazio (dado pelo tipo *void*) e a única função que parte de *void* para qualquer outro objeto é a *empty*, assim

$$true \circ empty = false \circ empty.$$

A função *empty* é um blefe e não pode ser chamada por causa da inexistência de argumentos. Assim, não há como fazer as duas constantes serem equalizadas. Não existir maneira de equalizar true e false é desejável, afinal não se espera que existam constantes repetidas ou redundantes numa linguagem de programação.

Definição 4.16. Coequalizador

Em um diagrama D de uma categoria **C**, composto par de objetos A, B e um par de morfismos paralelos $f, g : A \to B$, um equalizador de f, g é um cone $\langle P, \{p : B \to P\}\rangle$, tal que para qualquer cone $\langle K, \{k : B \to K\}\rangle$ existe um único morfismo $h : P \to K$, o qual

$$h \circ p = k.$$

4.10 Objeto Exponencial

Na categoria **Set** existem conjuntos e funções totais. Funções também formam conjuntos, conhecido como espaços de funções. Supondo dois conjuntos A e B finitos, a quantidade de mapeamentos $A \to B$ é dado pela exponenciação $|B|^{|A|}$ e espaço de função $\overrightarrow{B^A}$ é calculado por

$$\overrightarrow{B^A} = Hom(A, B)$$

Para que o conjunto potência exista como uma construção categórica é necessário representá-lo como uma relação comutativa de morfismos, sem utilizar a noção de elemento. Um objeto B^A deve ser tal que seu relacionamento com qualquer outro morfismo implique ser um objeto que representa o espaço de função $A \to B$.

Antes de generalizar exponenciação como uma construção categórica, será identificado como um conjunto potência se relaciona com os demais conjuntos e funções, para posteriormente abstrair o relacionamento comutativo. A generalização também precisa tratar de funções de múltiplos argumentos. Uma maneira de resolver isso é reescrever uma função binária (ou qualquer outra aridade maior que um) como uma função unária que retorna uma função. Ou ainda, como função unária que retorna um elemento de um espaço de função $\overrightarrow{B^A}$.

O esquema da Figura 4.11 mostra uma função $g : (C \times A) \to B$ e um espaço de função $\overrightarrow{B^A}$. A ideia é encontrar alguma função $A \to B$ que se comporte de maneira similar a g.

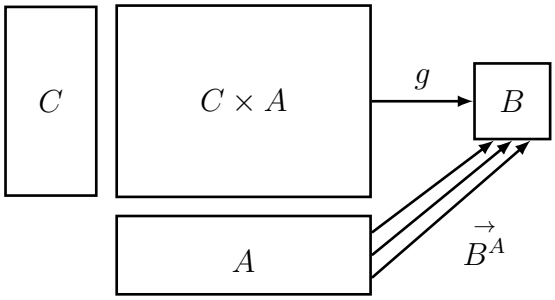

Figura 4.11: Esquema com função binária g e espaço de função B^A.

Suponha que o conjunto (objeto) B^A represente o espaço de funções $\overrightarrow{B^A}$, então deve ser o caso de existir uma função g_c, para algum $c \in C$, tal que para todos os $a \in A$

$$g_c(a) = g(c, a).$$

Então, g_c pode ser vista como g fixado em algum $c \in C$, que habita o espaço $\overrightarrow{B^A}$ e é um elemento de B^A. Um produto $B^A \times A$ representa uma função g_c acompanhada de um argumento a. Dessa maneira, uma função $eval_{B^A} : (B^A \times A) \to B$ definida por

$$eval_{B^A}(f, a) = f(a)$$

permite uma representação similar da Figura 4.11, como destaca a Figura 4.12.

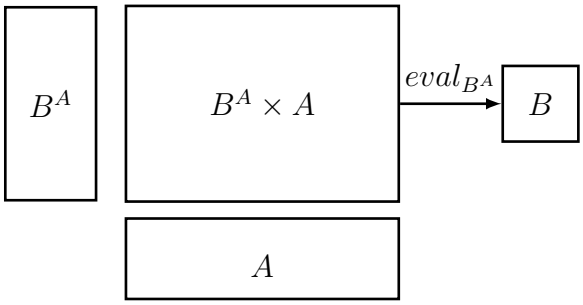

Figura 4.12: Esquema com função $eval_g$.

Ainda, precisa-se obter g_c a partir de g e $c \in C$. Uma maneira ingênua é definir g_c como uma aplicação parcial dos argumentos de g:

$$g_c = g(c),$$

porém não é possível admitir que g suporta aplicação parcial. Então, é necessário utilizar um truque. Cria-se uma função $eval_g : (((C \times A) \to B) \times C) \to B^A$ e uma função $\Lambda_g : C \to B^A$, tais que

$$eval_g(f, a) = f(a)$$
$$\Lambda_g(c) = eval_g(g, c) = g_c.$$

A função Λ_g é conhecida como currying e permite que g_c simule $g(c)$.

Agora, com $\Lambda_g : C \to B^A$ é possível criar uma relação entre as Figuras 4.11 e 4.12. A função

$$\Lambda_g \times Id_A : (C \times A) \to (B^A \times A)$$

pega g e um par de argumentos $(c, a) \in C \times A$ e leva à g_c e um argumento $a \in A$. Consequentemente, tem-se o seguinte diagrama.

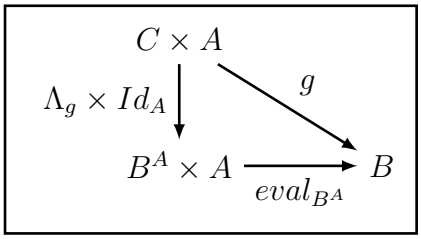

Figura 4.13: Diagrama do objeto exponencial.

Nem todos os morfismos $(C \times A) \to (B^A \times A)$ fazem o diagrama comutar.

Evidentemente $\Lambda_g \times Id_A$ faz, pois para qualquer $(c, a) \in C \times A$ tem-se

$$\begin{aligned} eval_{B^A} \circ \Lambda_g \times Id_A(c, a) &= eval_{B^A}(\Lambda_g(c), Id_A(a)) \\ &= eval_{B^A}(g_c, a) \\ &= g_c(a) = g(c, a). \end{aligned}$$

Ainda mais, Λ_g é única para $(C \times A) \to (B^A \times A)$ capaz de fazer o diagrama da Figura 4.13 comutar.

Prova: Suponha alguma função, diferente de Λ_g, $H : C \to B^A$, tal que para qualquer $(c, a) \in C \times A$

$$\begin{aligned} eval_{B^A} \circ H \times Id_A(c, a) &= g(c, a) \\ eval_{B^A}(H(c), Id_A(a)) &= g(c, a) \\ eval_{B^A}(H(c), a) &= g(c, a) \\ (H(c))(a) &= g(c, a). \end{aligned}$$

Não existe outro elemento além de g_c que satisfaça a igualdade, logo é necessário que

$$H(c) = \Lambda_g(c).$$

Nota 4.1. H e Λ_g apenas precisam ter o mesmo mapeamento. Se H e Λ_g tiverem regras de produção diferentes e o mesmo mapeamento, então H e Λ_g comutam o diagrama da Figura 4.13.

Definição 4.17. Objeto Exponencial

Seja uma categoria $\mathbf{C}$, ao qual para quaisquer pares de objetos $A, B \in Obj_C$ existe um objeto $A \times B \in Obj_{\mathbf{C}}$ do produto categorial. Um objeto exponencial é um par $\langle B^A, eval_{B^A} : B^A \times A \to B \rangle$ se para todo $C \in Obj_{\mathbf{C}}$ e $g : C \times A \to B$ existe um único morfismo $\Lambda_g : C \to B^A$ tal que

$$eval_{B^A} \circ \Lambda_g \times Id_A = g,$$

ou, então, o diagrama da Figura 6.2 comuta.

$$\begin{tikzcd} C \arrow[d, dashed, "\Lambda_g"] & C \times A \arrow[d, "\Lambda_g \times Id_A"'] \arrow[dr, "g"] & \\ B^A & B^A \times A \arrow[r, "eval_{B^A}"'] & B \end{tikzcd}$$

Figura 4.14: Diagrama comutativo do objeto exponencial.

Conclui-se que em uma categoria com produto e objeto exponencial definidos existe uma bijeção de coleções Hom para quaisquer objetos A, B e C (BERGER, 2010),

$$Hom(C \times A, B) \leftrightarrow Hom(C, B^A)$$

que implica na seguinte equivalência de objetos (ABRAMSKY; TZEVELEKOS, 2010):

$$B^{C \times A} \simeq (B^A)^C.$$

4.11 Categorias Cartesianas Fechadas

Uma Categoria Cartesiana Fechada, ou em inglês *Cartesian Closed Category* (CCC), é uma categoria especial, pois serve como uma interpretação semântica do Cálculo Lambda Tipado. Para qualquer $A, B \in Obj$ existe um $A \times B \in Obj$, e para qualquer morfismo $g : C \times A \to B$ há um objeto exponencial $\langle B^A, eval_{B^A} : B^A \times A \to B \rangle$.

O produto cartesiano pode ser interpretado como uma maneira de combinar objetos e formar produtos de aridade arbitrária. A exponenciação pode ser interpretada como a representação de morfismos por meio de objetos. Em particular, um objeto terminal 1 de uma categoria pode ser visto como um produto vazio () de aridade zero, ou como a como a exponenciação de um objeto A qualquer pelo objeto inicial 0, portanto,

$$1 \simeq A^0 \simeq ().$$

Em **Set**, qualquer conjunto de tamanho um é objeto terminal[1], que pode ser visto como o exponencial $A^\emptyset$ para qualquer A. Por fim, todos os produtos de aridade finita e maior, ou igual, a zero podem ser formados através do produto.

Definição 4.18. Categoria Cartesiana Fechada

Uma categoria **C** é Cartesiana Fechada se tem

(i) objeto terminal.

(ii) produto para qualquer par de objetos.

(iii) exponenciação para qualquer par de objetos.

[1]Pelo mesmo motivo de 1 ser o objeto terminal em **Prog**.

Categorias Cartesianas Fechadas são aquelas que podem definir transformações de aridade arbitrária, como a categoria **Set** e **Prog**, e permitem que morfismos sejam representados por objetos através da função *curry*.

5 Correspondência Curry-Howard-Lambek

A correspondência Curry-Howard-Lambek é uma tripla relação entre Cálculo Lambda Tipado, Dedução Natural Intuicionista e Categorias Cartesianas Fechadas. Inicialmente a correspondência era apenas o Isomorfismo de Curry-Howard (*Propositions as Types*) e foi, posteriormente, estendida por Lambek por meio de uma interpretação categórica (FERNANDES, 2011).

A descoberta dessa tripla relação resultou numa boa fundação para o desenvolvimento das linguagens de programação funcionais, em especial aquelas que tem polimorfismo paramétrico. Uma vez que proposições lógicas são sentenças que podem ser tautológicas, *Propositions as Types* mostra que isso é equivalente a tipos paramétricos habitados. Do lado categórico, linguagens funcionais têm tipos como objetos e funções como morfismos, o que pode resultar numa categoria cartesiana fechada.

Segundo (WADLER, 2015) *Propositions as Types* é uma noção com profundidade: vai na raiz das linguagens de programação funcional, explica funções, *records*, variantes, polimorfismo paramétrico, tipos de dados abstratos e *continuations*. Segundo o mesmo, também inspirou assistentes de provas como Coq e Agda.

Este capítulo define um fragmento da Dedução Natural Intuicionista, demostra o Isomorfismo Curry-Howard, define uma categoria que interpreta a Dedução Natural Intuicionista e, assim, estende a relação para Curry-Howard-Lambek.

5.1 Fragmento da Dedução Natural Intuicionista

Gerhard Gentzen por volta de 1930 criou um sistema de prova conhecido como Dedução Natural (GENTZEN, 1969), que utiliza provas formais definidas em árvores que são similares às apresentadas no Capítulo 3 de Sistemas de Tipos. 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 o termo $A \vee \neg A$, que significa A é verdadeiro ou $\neg A$ é verdadeiro, e pelo princípio do terceiro excluído conclui-se que o termo é sempre verdade

para quaisquer valores de 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) (MENEZES; HAEUSLER, 2001). 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;
- $A \rightarrow B$ é um procedimento tal que dadas construções de A, obtém-se construções de B;
- nada é uma construção de $\perp$. Assim, $\neg A$ é definido como $A \rightarrow \perp$, o que significa que $\neg A$ é um procedimento que gera a constante lógica sempre falso ($\perp$). Uma vez que $\perp$ 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. Uma noção muito similar a essa foi vista na Definição 2.26 de Tipo Habitável, em que para demonstrar a habitabilidade de um tipo τ era necessário apresentar um termo M e provar $M : \tau$.

O interesse desta seção é um fragmento da Dedução Natural Intuicionista, denotado por ND, que somente contém os conectivos conjunção $\wedge$ e implicação $\rightarrow$. A linguagem em questão é formalizada abaixo.

Definição 5.1. Proposição

Seja A um conjunto de símbolos chamados de átomos. O Conjunto P de proposições é definido indutivamente por:

[1]Prova em (MENEZES; HAEUSLER, 2001).

(i) A proposição $\top$ está em P.

(ii) Todo átomo é uma proposição em P.

(iii) Se τ e ϕ são proposições, então $(\tau \wedge \phi)$ é uma proposição e está em P.

(iv) Se τ e ϕ são proposições, então $(\tau \to \phi)$ é uma proposição e está em P.

Nota 5.1. O operador $\wedge$ tem maior precedência que $\to$, e $\to$ é associativo à direita, assim como tipo funcional.

Dedução Natural é uma forma de teoria da prova cujas regras de dedução são dadas aos pares (introdução e exclusão de conectivos lógicos). Além disso, as regras devem seguir a noção construtivista de prova que foi proposta por Brower, Heyting e Kolmogorov, conhecida com interpretação BHK, que são descritas por:

(i) Uma construção $(\tau \wedge \phi)$ é uma construção de τ e uma construção de ϕ.

(ii) Uma construção $(\tau \to \phi)$ é um procedimento que transforma toda construção de τ numa construção de ϕ.

(iii) Sempre é possível construir $\top$.

O conjunto Γ é chamado de contexto, assim como nos modelos tipados de Cálculo Lambda, e seus elementos são proposições do conjunto P conhecidos como hipóteses. Uma expressão da forma $\Gamma \vdash \tau$ é chamada de sequente e denota que τ é deduzido a partir do contexto Γ. Uma prova é um sequente cujo contexto é o conjunto vazio. Uma árvore de prova (*proof tree*) é uma árvore composta por sequentes, regras de dedução e uma raiz de forma $\emptyset \vdash \tau$.

Definição 5.2. Regras da Dedução Natural Intuicionista

$$\frac{}{\Gamma \cup \{\tau\} \vdash \tau}\,(ax) \qquad \frac{}{\Gamma \vdash \top}\,(\top)$$

$$\frac{\Gamma_1 \vdash \alpha \to \tau \qquad \Gamma_2 \vdash \alpha}{\Gamma_1 \cup \Gamma_2 \vdash \tau}\,(\to_E) \qquad \frac{\Gamma \vdash \tau}{\Gamma \setminus \{\alpha\} \vdash \alpha \to \tau}\,(\to_I)$$

$$\frac{\Gamma_1 \vdash \alpha \qquad \Gamma_2 \vdash \tau}{\Gamma_1 \cup \Gamma_2 \vdash \alpha \wedge \tau}\,(\wedge_I) \qquad \frac{\Gamma \vdash \alpha \wedge \tau}{\Gamma \vdash \alpha}\,(\wedge_{E_1}) \qquad \frac{\Gamma \vdash \alpha \wedge \tau}{\Gamma \vdash \tau}\,(\wedge_{E_2})$$

Algumas verdades lógicas são provadas abaixo. Uma proposição sempre implica nela mesma (5.1a), se τ é assumido como verdade e σ é assumido como verdade, então conclui-se τ. (5.1b). Se é possível assumir σ e assumir τ, então é possível provar $\sigma \wedge \tau$ (5.1c). Note as similaridades destas provas com as do Sistema TA (Exemplo 2.3). Isso é o suficiente para dar uma noção intuitiva de termos como provas e tipos como proposições (*propositions as types*), ou também chamado de Isomorfismo de Curry-Howard.

Exemplo 5.1. Provas em Dedução Natural Intuicionista

$$\text{(a)} \quad \dfrac{\sigma \vdash \sigma}{\vdash \sigma \to \sigma}\,(\to_I)$$

$$\text{(b)} \quad \dfrac{\dfrac{\tau \vdash \tau}{\tau \vdash \sigma \to \tau}\,(\to_I)}{\vdash \tau \to \sigma \to \tau}\,(\to_I)$$

$$\text{(c)} \quad \dfrac{\dfrac{\dfrac{\sigma \vdash \sigma \qquad \tau \vdash \tau}{\sigma, \tau \vdash \sigma \wedge \sigma}\,(\wedge_I)}{\vdash \tau \to \sigma \wedge \tau}\,(\to_I)}{\vdash \tau \to \sigma \to \sigma \wedge \tau}\,(\to_I)$$

Apesar de isomorfismo não ter sido definido por meio de Teoria das Categorias, a ideia intuitiva é suficiente. Isomorfismo é uma relação bijetiva (ida e volta) entre duas estruturas matemáticas, onde suas relações internas são preservadas pelo mapeamento. Por exemplo, o isomorfismo de grafos é um mapeamento de nós e arestas e que, também, preserva a relação de adjacência dos nós. Portanto, o último requisito, para começar a provar o isomorfismo, é criar uma relação interna, denotada por $\rightsquigarrow$, entre os objetos da Dedução Natural e que seja análoga às regras de Redução-β do Cálculo Lambda.

Definição 5.3. Regras de Simplificação

(i) Simplificação $\wedge_1$:

$$\dfrac{\dfrac{\dfrac{\Sigma}{\Gamma_1 \vdash \varphi} \qquad \dfrac{\Pi}{\Gamma_2 \vdash \phi}}{\Gamma_1 \cup \Gamma_2 \vdash \varphi \wedge \phi}\,\wedge_I}{\Gamma_1 \vdash \varphi}\,\wedge_{E_1} \quad \rightsquigarrow \quad \dfrac{\Sigma}{\Gamma_1 \vdash \varphi}$$

(ii) Simplificação $\wedge_2$:

$$\dfrac{\dfrac{\dfrac{\Sigma}{\Gamma_1 \vdash \varphi} \qquad \dfrac{\Pi}{\Gamma_2 \vdash \phi}}{\Gamma_1 \cup \Gamma_2 \vdash \varphi \wedge \phi}\,\wedge_I}{\Gamma_2 \vdash \phi}\,\wedge_{E_2} \quad \rightsquigarrow \quad \dfrac{\Pi}{\Gamma_2 \vdash \phi}$$

(iii) Simplificação $\rightarrow$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\phi\} \vdash \phi}{\Sigma}}{\Gamma_1 \cup \{\phi\} \vdash \varphi}}{\Gamma_1 \vdash \phi \rightarrow \varphi} \rightarrow_I \quad \dfrac{\Pi}{\Gamma_2 \vdash \phi}}{\Gamma_1 \cup \Gamma_2 \vdash \varphi} \rightarrow_E \quad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{\Pi}{\Gamma_2 \vdash \phi}}{\Sigma}}{\Gamma_1 \cup \Gamma_2 \vdash \varphi}$$

Onde Σ e Π são quaisquer árvores de derivação.

As duas primeiras regras afirmam que se for possível provar φ, ou ϕ em outra derivação, e posteriormente for utilizada a regra $\wedge_I$ para inferir $\varphi \wedge \phi$ e, depois, $\wedge_{E_1}$ ou $\wedge_{E_2}$, então é possível simplificar para apenas as provas de φ ou ϕ. A terceira regra endossa que se for provado φ a partir da suposição de ϕ e posteriormente for utilizado a regra $\rightarrow_I$ para conseguir uma prova de $\varphi \rightarrow \phi$. Se, também, houver uma prova de ϕ, ao invés de utilizar a regra $\rightarrow_E$, pode-se substituir a suposição de ϕ, na prova de φ, pela prova de ϕ.

Lema 5.1. Lema da Substituição

Se Π for uma prova de $\Gamma \cup \{\phi\} \vdash \varphi$ e Σ, uma prova de $\Gamma \vdash \phi$, então existe uma prova $\mathbb{S}(\Pi, \Sigma)$ de $\Gamma \vdash \varphi$. A prova deste lema é uma indução estruturada e esta presente em (FERNANDES, 2011).

5.2 Isomorfismo Curry-Howard

A prova do Isomorfismo de Curry-Howard é uma indução estruturada. Divide-se a prova em duas partes: a primeira relaciona tipos com proposições e termos com provas, e a segunda relaciona normalização de termos com simplificação de provas.

O primeiro passo é mostrar que não há distinção entre proposições e tipos. Para isso, considere como iguais o conjunto de variáveis de tipo $\mathbb{W}$ do sistema Type Assignment (TA) e conjunto de átomos P do fragmento da Dedução Intuicionista (ND), então define-se a função, trivialmente bijetiva, $H : \text{ND} \rightarrow \text{TA}$, tal que

$$\begin{aligned} H(\top) &= 1, \\ H(\sigma) &= \sigma, \ se\ \sigma \in P, \\ H(\sigma \rightarrow \tau) &= \sigma \rightarrow \tau, \\ H(\sigma \wedge \tau) &= \sigma \times \tau. \end{aligned}$$

Também é necessário conseguir transformar o contexto do sistema TA no da Dedução Natural e vice-versa. Isso é feito, respectivamente, com as seguintes duas funções:

$$TVars(\Gamma) = \{H^{-1}(\tau) \mid (x : \tau) \in \Gamma\};$$
$$NVars(\Gamma) = \{x_\rho : H(\rho) \mid \rho \in \Gamma\}.$$

Teorema 5.1. Isomorfismo de Curry-Howard Parte I

(i) Se (em TA) $\Gamma \vdash M : \tau$, então (em ND) $TVars(\Gamma) \vdash H^{-1}(\tau)$.

(ii) Se (em ND) $\Gamma \vdash \tau$, então (em TA) existe um termo M, tal que $NVars(\Gamma) \vdash M : H(\tau)$.

Prova 5.1. Isomorfismo de Curry-Howard Parte I

(i) Indução caso a caso das fórmulas $\Gamma \vdash M : \tau$ do sistema TA:

- Caso (base) $M \equiv x$:
  Tendo-se $\Gamma \vdash x : \tau$, então $x : \tau \in \Gamma$, pois x é variável e, portanto, a prova vem do uso da regra (var):

  $$\frac{}{x : \tau \vdash x : \tau}\ (var)$$

  logo, aplicando-se as funções de conversão tem-se

  $$TVars(\{x : \tau\}) \vdash H^{-1}(\tau)$$

  o que é válido em ND. Mais ainda, para qualquer Γ, tal que $x : \tau \in \Gamma$, tem-se

  $$TVars(\Gamma) \vdash H^{-1}(\tau).$$

- Caso (base) $M \equiv *$:
  Tendo-se $\Gamma \vdash * : \tau$, então pela regra $(unit)$ é necessário que $\tau = 1$. Aplicando-se as funções de conversão tem-se

  $$TVars(\Gamma) \vdash H^{-1}(1).$$

- Caso (passo) $M \equiv \lambda x.N$:
  A hipótese

  $$\Gamma' \vdash N : \tau \Rightarrow TVars(\Gamma') \vdash H^{-1}(\tau)$$

deve permitir provar o caso $M \equiv \lambda x.N$, ou seja, a partir de

$$\Gamma \setminus x \vdash \lambda x.N : \alpha \to \tau$$

provar

$$TVars(\Gamma \setminus \alpha) \vdash H^{-1}(\alpha \to \tau).$$

Assim, supondo uma derivação de $\Gamma \setminus x \vdash \lambda x.N : \alpha \to \tau$ em TA, que necessariamente foi obtida pela aplicação da regra (abs) a partir de

$$\Gamma \vdash N : \tau.$$

Logo, aplicando a hipótese de indução nessa última derivação, tem-se a derivação em ND de

$$TVars(\Gamma) \vdash H^{-1}(\tau)$$

e a partir disso, aplica-se a regra $(\to_I)$ resultando em

$$TVars(\Gamma \setminus \alpha) \vdash H^{-1}(\alpha \to \tau).$$

- Caso (passo) $M \equiv PQ$:
  As hipóteses

$$\Gamma'_1 \vdash P : \alpha \to \tau \Rightarrow TVars(\Gamma'_1) \vdash H^{-1}(\alpha \to \tau);$$
$$\Gamma'_2 \vdash Q : \alpha \Rightarrow TVars(\Gamma'_2) \vdash H^{-1}(\alpha)$$

devem permitir provar o caso $M \equiv PQ$, ou seja, a partir de

$$\Gamma_1 \cup \Gamma_2 \vdash (PQ) : \tau$$

provar

$$TVars(\Gamma_1 \cup \Gamma_2) \vdash H^{-1}(\tau).$$

Assim, supondo uma derivação de $\Gamma_1 \cup \Gamma_2 \vdash (PQ) : \tau$ em TA, que impreterivelmente foi obtida pela aplicação da regra (app) a partir de

$$\Gamma_1 \vdash P : \alpha \to \tau;$$
$$\Gamma_2 \vdash Q : \alpha.$$

Logo, aplicando a hipótese de indução nessas últimas derivações, tem-se as derivações em ND de

$$TVars(\Gamma_1) \vdash H^{-1}(\alpha \to \tau);$$
$$TVars(\Gamma_2) \vdash H^{-1}(\alpha).$$

e a partir disso, aplica-se a regra $(\to_E)$ resultando em

$$TVars(\Gamma_1 \cup \Gamma_2) \vdash H^{-1}(\tau).$$

- Caso (passo) $M \equiv \langle P, Q \rangle$:
  As hipóteses

$$\Gamma'_1 \vdash P : \alpha \Rightarrow TVars(\Gamma'_1) \vdash H^{-1}(\alpha);$$
$$\Gamma'_2 \vdash Q : \tau \Rightarrow TVars(\Gamma'_2) \vdash H^{-1}(\tau)$$

  devem permitir provar o caso $M \equiv \langle P, Q \rangle$, ou seja, a partir de

$$\Gamma_1 \cup \Gamma_2 \vdash \langle P, Q \rangle : \alpha \times \tau$$

  provar

$$TVars(\Gamma_1 \cup \Gamma_2) \vdash H^{-1}(\alpha \times \tau).$$

  Assim, supondo uma derivação de $\Gamma_1 \cup \Gamma_2 \vdash \langle P, Q \rangle : \alpha \times \tau$ em TA, que necessariamente foi obtida pela aplicação da regra (*pair*) a partir de

$$\Gamma_1 \vdash P : \alpha;$$
$$\Gamma_2 \vdash Q : \tau.$$

  Logo, aplicando a hipótese de indução nessas últimas derivações, tem-se as derivações em ND de

$$TVars(\Gamma_1) \vdash H^{-1}(\alpha);$$
$$TVars(\Gamma_2) \vdash H^{-1}(\tau).$$

  e a partir disso, aplica-se a regra $(\wedge_I)$ resultando em

$$TVars(\Gamma_1 \cup \Gamma_2) \vdash H^{-1}(\alpha \times \tau).$$

- Caso (passo) $M \equiv \pi_1 N$:

  A hipótese

$$\Gamma' \vdash N : \alpha \times \tau \Rightarrow TVars(\Gamma') \vdash H^{-1}(\alpha \times \tau)$$

  deve permitir provar o caso $M \equiv \pi_1 N$, ou seja, a partir de

$$\Gamma \vdash \pi_1 N : \alpha$$

  provar

$$TVars(\Gamma) \vdash H^{-1}(\alpha).$$

  Assim, supondo uma derivação de $\Gamma \vdash \pi_1 N : \alpha$ em TA, que necessariamente foi obtida pela aplicação da regra (π_1) a partir de

$$\Gamma \vdash N : \alpha \times \tau$$

  Logo, aplicando a hipótese de indução nessas últimas derivações, tem-se as derivações em ND de

$$TVars(\Gamma) \vdash H^{-1}(\alpha \times \tau)$$

  e a partir disso, aplica-se a regra $(\wedge_{E_1})$ resultando em

$$TVars(\Gamma) \vdash H^{-1}(\alpha).$$

- Caso (passo) $M \equiv \pi_2 N$:

  A hipótese

$$\Gamma' \vdash N : \alpha \times \tau \Rightarrow TVars(\Gamma') \vdash H^{-1}(\alpha \times \tau)$$

  deve permitir provar o caso $M \equiv \pi_2 N$, ou seja, a partir de

$$\Gamma \vdash \pi_2 N : \tau$$

  provar

$$TVars(\Gamma) \vdash H^{-1}(\tau).$$

  Assim, supondo uma derivação de $\Gamma \vdash \pi_2 N : \tau$ em TA, que necessariamente foi obtida pela aplicação da regra (π_2) a partir de

$$\Gamma \vdash N : \alpha \times \tau$$

Logo, aplicando a hipótese de indução nessas últimas derivações, tem-se as derivações em ND de

$$TVars(\Gamma) \vdash H^{-1}(\alpha \times \tau)$$

e a partir disso, aplica-se a regra $(\wedge_{E_2})$ resultando em

$$TVars(\Gamma) \vdash H^{-1}(\tau).$$

(ii) Indução caso a caso nas provas $\Gamma \vdash \tau$ em ND:

- Caso (base) da regra (ax):
  Tendo-se $\Gamma \vdash \tau$ e $\tau \in \Gamma$, então a regra (ax):

  $$\frac{}{\Gamma \cup \{\tau\} \vdash \tau}\,(ax)$$

  foi utilizada para derivar essa fórmula. Assim, aplicando as funções de conversão tem-se

  $$NVars(\Gamma) \vdash x_\tau : H(\tau)$$

  o que é uma fórmula válida em TA, pois pela definição de $NVars$

  $$\tau \in \Gamma \Rightarrow x_\tau : H(\tau) \in NVars(\Gamma).$$

- Caso (base) da regra $(\top)$:
  Tendo-se $\Gamma \vdash \top$, então a regra $(\top)$ necessariamente foi utilizada, assim aplicando-se as funções de conversão tem-se

  $$NVars(\Gamma) \vdash * : H(1).$$

- Caso (passo) da regra $(\rightarrow_I)$:
  A hipótese

  $$\Gamma' \vdash \tau \Rightarrow NVars(\Gamma') \vdash N : H(\tau)$$

  deve permitir provar o caso da regra $(\rightarrow_I)$, ou seja, a partir de

  $$\Gamma \setminus \alpha \vdash \alpha \rightarrow \tau$$

provar

$$NVars(\Gamma \setminus \alpha) \vdash \lambda x.N : H(\alpha \to \tau).$$

Assim, supondo uma derivação $\Gamma \setminus \alpha \vdash \alpha \to \tau$ em ND, a qual necessariamente foi obtida pela aplicação da regra $(\to_I)$ a partir de

$$\Gamma \vdash \tau.$$

Logo, aplicando a hipótese de indução nesta última derivação, tem-se a derivação em TA de

$$NVars(\Gamma) \vdash N : H(\tau)$$

e a partir disso, aplica-se a regra (abs) resultando em

$$NVars(\Gamma \setminus \alpha) \vdash \lambda x.N : H(\alpha \to \tau).$$

- Caso (passo) da regra $(\to_E)$:
  As hipóteses

$$\Gamma'_1 \vdash \alpha \to \tau \Rightarrow NVars(\Gamma'_1) \vdash P : H(\alpha \to \tau);$$
$$\Gamma'_2 \vdash \alpha \Rightarrow NVars(\Gamma'_2) \vdash Q : H(\alpha)$$

  devem permitir provar o caso da regra $(\to_E)$, ou seja, a partir de

$$\Gamma_1 \cup \Gamma_2 \vdash \tau$$

  provar

$$NVars(\Gamma_1 \cup \Gamma_2) \vdash PQ : H(\tau).$$

  Assim, supondo uma derivação $\Gamma_1 \cup \Gamma_2 \vdash \tau$ em ND, que foi obtida pela aplicação da regra $(\to_E)$ a partir de

$$\Gamma_1 \vdash \alpha \to \tau;$$
$$\Gamma_2 \vdash \alpha.$$

  Logo, aplicando a hipótese de indução nestas últimas derivações, tem-se a derivação em TA de

$$NVars(\Gamma_1) \vdash P : H(\alpha \to \tau);$$
$$NVars(\Gamma_2) \vdash Q : H(\alpha).$$

e a partir disso, usa-se a regra (*app*) resultando em

$$NVars(\Gamma_1 \cup \Gamma_2) \vdash PQ : H(\tau).$$

- Caso (passo) da regra $(\wedge_I)$:

  As hipóteses

$$\Gamma'_1 \vdash \alpha \Rightarrow NVars(\Gamma'_1) \vdash P : H(\alpha);$$
$$\Gamma'_2 \vdash \tau \Rightarrow NVars(\Gamma'_2) \vdash Q : H(\tau)$$

  devem permitir provar o caso da regra $(\wedge_I)$, ou seja, a partir de

$$\Gamma_1 \cup \Gamma_2 \vdash \alpha \wedge \tau$$

  provar

$$NVars(\Gamma_1 \cup \Gamma_2) \vdash \langle P, Q\rangle : H(\alpha \wedge \tau)$$

  Assim, supondo uma derivação $\Gamma_1 \cup \Gamma_2 \vdash \alpha \wedge \tau$ em ND, que foi obtida pela aplicação da regra $(\wedge_I)$ a partir de

$$\Gamma'_1 \vdash \alpha;$$
$$\Gamma'_2 \vdash \tau$$

  Logo, aplicando a hipótese de indução nestas últimas derivações, tem-se a derivação em TA de

$$NVars(\Gamma'_1) \vdash P : (\alpha);$$
$$NVars(\Gamma'_2) \vdash Q : (\tau).$$

  e a partir disso, aplica-se a regra (*pair*) resultando em

$$NVars(\Gamma_1 \cup \Gamma_2) \vdash \langle P, Q\rangle : H(\alpha \wedge \tau).$$

- Caso (passo) da regra $(\wedge_{E_1})$:

  A hipótese

$$\Gamma' \vdash \alpha \Rightarrow NVars(\Gamma') \vdash \pi_1 N : H(\alpha)$$

  deve permitir provar o caso da regra $(\wedge_{E_1})$, ou seja, a partir de

$$\Gamma \vdash \alpha$$

provar

$$NVars(\Gamma) \vdash \pi_1 N : H(\alpha)$$

Assim, supondo uma derivação $\Gamma \vdash \alpha$ em ND, que foi obtida pela aplicação da regra $(\wedge_{E_1})$ a partir de

$$\Gamma \vdash \alpha \wedge \tau;$$

Logo, aplicando a hipótese de indução nesta última derivação, tem-se a derivação em TA de

$$NVars(\Gamma) \vdash N : H(\alpha \wedge \tau)$$

e a partir disso, aplica-se a regra (π_1) resultando em

$$NVars(\Gamma) \vdash \pi_1 N : H(\alpha).$$

- Caso (passo) da regra $(\wedge_{E_2})$:
  A hipótese

$$\Gamma' \vdash \tau \Rightarrow NVars(\Gamma') \vdash \pi_2 N : H(\tau)$$

deve permitir provar o caso da regra $(\wedge_{E_2})$, ou seja, a partir de

$$\Gamma \vdash \tau$$

provar

$$NVars(\Gamma) \vdash \pi_2 N : H(\tau)$$

Assim, supondo uma derivação $\Gamma \vdash \tau$ em ND, que foi obtida pela aplicação da regra $(\wedge_{E_2})$ a partir de

$$\Gamma \vdash \alpha \wedge \tau;$$

Logo, aplicando a hipótese de indução nesta última derivação, tem-se a derivação em TA de

$$NVars(\Gamma) \vdash N : H(\alpha \wedge \tau)$$

e a partir disso, aplica-se a regra (π_2) resultando em

$$NVars(\Gamma) \vdash \pi_2 N : H(\tau).$$

Teorema 5.2. Isomorfismo de Curry-Howard Parte II

Sejam D e D' árvores de derivações em ND, determinadas pelos termos M e M' em TA, respectivamente.

(i) Se (em TA) $M \to_\beta M'$, então (em ND) $D \rightsquigarrow D'$.

(ii) Se (em ND) $D \rightsquigarrow D'$, então (em TA) $M \to_\beta M'$.

Prova 5.2. Isomorfismo de Curry-Howard Parte II

(i) Por indução da relação $M \to M'$:

- Caso $M \equiv (\lambda x.M_1)M_2 : \tau$ e $M' \equiv [M_2/x]M_1 : \tau$, então pelo Teorema 5.1 o termo M corresponde à derivação D

$$\dfrac{\dfrac{\dfrac{\Sigma}{\Gamma \vdash \tau}}{\Gamma \setminus \phi \vdash \alpha \to \tau} \to_I \quad \dfrac{\Pi}{\Gamma \vdash \alpha}}{\Gamma \vdash \tau} \to_E$$

e pela simplificação $\to$ o termo M' corresponde à derivação D'

$$\dfrac{\dfrac{\Pi}{\Gamma \vdash \alpha}}{\dfrac{\Sigma}{\Gamma \vdash \tau}}$$

- Caso $M \equiv \pi_1\langle M_1, M_2\rangle : \tau$ e $M' \equiv M_1 : \tau$, então pelo Teorema 5.1 o termo M corresponde à derivação D

$$\dfrac{\dfrac{\dfrac{\Sigma}{\Gamma \vdash \tau} \quad \dfrac{\Pi}{\Gamma \vdash \alpha}}{\Gamma \vdash \tau \wedge \alpha} \wedge_I}{\Gamma \vdash \tau} \wedge_{E_1}$$

e pela simplificação $\wedge_1$ o termo M' corresponde à derivação D'

$$\dfrac{\Sigma}{\Gamma \vdash \tau}$$

- Caso $M \equiv \pi_2\langle M_1, M_2\rangle : \alpha$ e $M' \equiv M_2 : \alpha$, então pelo Teorema 5.1 o termo M corresponde à derivação D

$$\dfrac{\dfrac{\dfrac{\Sigma}{\Gamma \vdash \tau} \quad \dfrac{\Pi}{\Gamma \vdash \alpha}}{\Gamma \vdash \tau \wedge \alpha} \wedge_I}{\Gamma \vdash \alpha} \wedge_{E_2}$$

e pela simplificação $\wedge_2$ o termo M' corresponde à derivação D'

$$\dfrac{\Pi}{\Gamma \vdash \alpha}$$

(ii) Por indução da relação $D \rightsquigarrow D'$:

- Caso D seja a derivação

$$\dfrac{\dfrac{\dfrac{\Sigma}{\Gamma \vdash \tau}}{\Gamma \setminus \phi \vdash \alpha \to \tau} \to_I \qquad \dfrac{\Pi}{\Gamma \vdash \alpha}}{\Gamma \vdash \tau} \to_E$$

  a qual pelo Teorema 5.1 corresponde ao termo $M \equiv (\lambda x.M_1)M_2 : \tau$. Assim, pela simplificação $\to$ há um D', tal que $D \rightsquigarrow D'$, que corresponde ao termo $[M_2/x]M_1 : \tau$ (também pelo Teorema 5.1).

- Caso D seja a derivação

$$\dfrac{\dfrac{\dfrac{\Sigma}{\Gamma \vdash \tau} \qquad \dfrac{\Pi}{\Gamma \vdash \alpha}}{\Gamma \vdash \tau \wedge \alpha} \wedge_I}{\Gamma \vdash \tau} \wedge_{E_1}$$

  a qual pelo Teorema 5.1 corresponde ao termo $M \equiv \pi_1\langle M_1, M_2\rangle : \tau$. Assim, pela simplificação $\wedge_1$ há um D', tal que $D \rightsquigarrow D'$, que corresponde ao termo $M_1 : \tau$ (também pelo Teorema 5.1).

- Caso D seja a derivação

$$\dfrac{\dfrac{\dfrac{\Sigma}{\Gamma \vdash \tau} \qquad \dfrac{\Pi}{\Gamma \vdash \alpha}}{\Gamma \vdash \tau \wedge \alpha} \wedge_I}{\Gamma \vdash \alpha} \wedge_{E_2}$$

  a qual pelo Teorema 5.1 corresponde ao termo $M \equiv \pi_2\langle M_1, M_2\rangle : \alpha$. Assim, pela simplificação $\wedge_2$ há um D', tal que $D \rightsquigarrow D'$, que corresponde ao termo $M_2 : \alpha$ (também pelo Teorema 5.1).

Agora que a relação isomorfa foi comprovada, é factível comentar o que ela significa e suas consequências. A semântica BHK da lógica intuicionista está relacionada com o papel do Cálculo Lambda como uma teoria computacional e uma teoria sobre funções. Em especial, o significado da implicação é um procedimento que transforma uma construção, o que é precisamente capturado pela abstração λ.

Considere o problema *inhabitation*: Dado um tipo τ, existe um termo M, tal que $\vDash M : \tau$? Pelo Teorema 5.1, é fácil perceber que isso pode ser traduzido para Dedução Natural como: Dado uma proposição τ, existe uma derivação D, cuja raiz é $\vdash \tau$?

Em resumo, a correspondência *Propositions as Types* é:

$$\text{Fórmulas} \leftrightarrow \text{Tipos}$$

$$\text{Provas} \leftrightarrow \text{Termos}$$

$$\text{Redução-}\beta \leftrightarrow \text{Simplificação de provas}$$

$$\text{Provabilidade} \leftrightarrow \textit{Inhabitation}$$

$$\text{Teorema} \leftrightarrow \text{Tipo habitado}$$

5.3 Curry-Howard-Lambek

Esta seção é sobre a semântica categórica do sistema TA e do sistema ND, ou seja, sobre como ambos sistemas são CCC. A prova original da tripla relação CHL é demostrada por Lambek em (LAMBEK, 1986), e também está presente em (LAMBEK; SCOTT, 1988), (BELL, 2014) e (BERGER, 2010). Nesses, define-se uma categoria cujos objetos são Cálculos Lambda Tipados e, então, é provado uma relação isomorfa com a categoria de todas as categorias cartesianas fechadas. Portanto, qualquer teoria de um Cálculo Lambda Tipado determina uma CCC e, inversamente, uma CCC qualquer gera um Cálculo Lambda Tipado. Esse resultado geral implica que, possivelmente, os demais sistemas de tipos do Capítulo 3 também são CCC.

Uma maneira simples, porém mais limitada, de mostrar a interpretação categórica é provar que o sistema ND (ou o sistema TA) tem a mesma semântica que uma Categoria Cartesiana Fechada. Para isso, basta definir uma categoria que representa o sistema ND e, então, provar que essa é Cartesiana Fechada. Como foi já foi demonstrado que TA e ND são isomorfos, bastaria apenas demostrar um dos lados. Porém, o que se segue nesta seção é a demostração da semântica categórica tando do sistema ND quanto do sistema TA.

5.3.1 Dedução Natural Intuicionista é uma Categoria Cartesiana Fechada

A ideia central da semântica categórica do sistema ND é interpretar proposições como objetos e sequentes (provas) como morfismos. Os passos que permitem essa interpretação são: definir uma relação de pré-ordem $\leq$ com as proposições de ND, definir uma classe

de equivalência $\bar{P}$ com as proposições de ND e, então, definir a categoria de ND a partir do conjunto parcialmente ordenado $(\bar{P}, \leq)$, que devido à sua unicidade de morfismos irá facilitar a posterior identificação como categoria cartesiana fechada.

Nota 5.2. No sistema ND, o que está do lado esquerdo do $\vdash$ é conhecido como contexto e é um conjunto de proposições. Para que $\vdash$ seja reconhecido como uma relação $\leq$ entre proposições, então converte-se contexto $\{\phi_1, \phi_2, .., \phi_n\}$ para a conjunção $\phi_1 \wedge \phi_2 \wedge ... \wedge \phi_n$ e caso $n = 0$, então $\top$.

Nota 5.3. Um sequente $\Gamma \vdash \varphi$ pode ser interpretado como uma prova $\emptyset \vdash \Gamma \to \varphi$ e vice-versa. Pois, se o contexto Γ coexiste com a provada proposição φ, então Γ implica em φ. Inclusive, essa é a essência das regras $(\to_I)$ e (abs). Por isso, pelo que se segue, provas e sequentes têm o mesmo significado.

A relação de pré-ordem $\leq$ é definida por

$$\varphi \leq \phi \Longleftrightarrow \text{existe uma prova de } \varphi \vdash \phi.$$

A reflexividade de $\leq$ é garantida pela regra de derivação (ax), que permite provar para qualquer proposição ϕ que $\phi \vdash \phi$. A transitividade $\phi \leq \varphi$ é provada pelo Lema 5.1 da Substituição de provas:

$$\text{se} \quad \frac{\Sigma}{\Gamma \vdash \phi} \quad \text{e} \quad \frac{\Pi}{\Gamma \cup \{\phi\} \vdash \varphi} \quad \text{então} \quad \frac{\mathbb{S}(\Pi, \Sigma)}{\Gamma \vdash \varphi}.$$

A antisimetria não é garantida, pois é possível $\varphi \vdash \phi$ e $\phi \vdash \varphi$ sem que $\phi = \varphi$, basta $\varphi = v$ e $\phi = v \wedge v$. Como se visa criar um conjunto parcialmente ordenado com as proposições de ND, então é preciso criar uma relação de equivalência, denotada por $\simeq$, que seja menos estrita que a igualdade sintática de proposições $=$. Define-se $\simeq$ por:

$$\varphi \simeq \phi \Longleftrightarrow \text{ se há provas de } \varphi \vdash \phi \text{ e } \phi \vdash \varphi.$$

Nota 5.4. Observe que $v \simeq v \wedge v$ e por isso garantirá a antisimetria.

Ao invés de utilizar o conjunto P de proposições de ND, será definido um conjunto $\bar{P}$ cujos elementos são classes de equivalência. Uma classe de equivalência de uma proposição ϕ é

$$[\phi] = \{\varphi \in P \mid \varphi \simeq \phi\}$$

e o conjunto $\bar{P}$ é definido por

$$\bar{P} = \{[\varphi] \mid \varphi \in P\}.$$

Uma classe de equivalência notável é $[\top] = \{\top, \tau \to \tau ...\}$, que tem todos os teoremas. Veja que por definição $\vdash \tau \to \tau$ é $\top \leq \tau \to \tau$, pois afinal é o que se espera de um sistema lógico consistente: o valor verdade de uma proposição é preservado pela consequência lógica.

Conclui-se que $(\bar{P}, \leq)$ é um conjunto parcialmente ordenado (reflexivo, transitivo e antisimétrico). Agora é possível definir a categoria ND, denotada por **CND**, da seguinte maneira:

Definição 5.4. Categoria **CND**

(a) Obj_{CND} é o conjunto $\bar{P}$, ou seja, os objetos são as classes de equivalência.

(b) Mor_{CND} é definido pela relação $\leq$. Assim, para quaisquer $\varphi, \phi \in \bar{P}$ existe uma seta/morfismo $f : \phi \to \varphi$ se, e somente se, a relação $\phi \leq \varphi$ é verdadeira.
Como morfismos são gerados por provas, então são obtidos pelas seguintes regras da Dedução Natural adaptadas:

$$\frac{}{f : \Gamma \wedge \tau \to \tau}\,(ax) \qquad \frac{}{\bigcirc_\Gamma : \Gamma \to \top}\,(\top)$$

$$\frac{f : \Gamma_1 \to \alpha \to \tau \qquad g : \Gamma_2 \to \alpha}{eval_{\phi \to \varphi} \circ \langle f, g \rangle : \Gamma_1 \wedge \Gamma_2 \to \tau}\,(\to_E) \qquad \frac{g : \Gamma \wedge \alpha \to \tau}{\Lambda_g : \Gamma \to (\alpha \to \tau)}\,(\to_I)$$

$$\frac{f : \Gamma_1 \to \alpha \qquad g : \Gamma_2 \to \tau}{\langle f, g \rangle : \Gamma_1 \wedge \Gamma_2 \to \alpha \wedge \tau}\,(\wedge_I)$$

$$\frac{f : \Gamma \to \alpha \wedge \tau}{\pi_1 \circ f : \Gamma \to \alpha}\,(\wedge_{E_1}) \qquad \frac{f : \Gamma \to \alpha \wedge \tau}{\pi_2 \circ f : \Gamma \to \tau}\,(\wedge_{E_2})$$

(c) ∂_0 e ∂_1 são as funções que levam, respectivamente, uma relação à sua classe de equivalência de origem e destino.

(d) $\circ$ é a composição da relação $\leq$ e é provada pela transitividade da mesma. Portanto, se $f : \Gamma \leq \varphi$ e $g : \varphi \to \phi$, então $g \circ f : \Gamma \to \phi$.

(e) ι retorna a relação reflexiva $\varphi \vdash \varphi$ de cada classe de equivalência $[\varphi]$.

Nota 5.5. Os símbolos $\vdash$ e $\leq$ quando tratados como morfismos são representados por $\rightarrow$. A implicação lógica $\rightarrow$ do sistema ND também é representada pelo mesmo símbolo, pois não há diferença entre uma prova de $\phi \rightarrow \phi \vdash \phi \rightarrow \phi$ e o morfismo $f : (\phi \rightarrow \phi) \rightarrow \phi \rightarrow \phi$.

Alguns morfismos especiais são apresentados para auxiliar na demostração de que **CND** é CCC. Se D_1 e D_2 são provas, respectivamente, de $\Gamma_2 \vdash \varphi$ e $\Gamma_1 \vdash \phi$, então pela regra $(\rightarrow_I)$ $\Gamma_1 \cup \Gamma_2 \vdash \varphi \wedge \phi$. Se $f : \Gamma_1 \rightarrow \varphi$ e $g : \Gamma_2 \rightarrow \phi$ são setas que representam, respectivamente, D_1 e D_2, então existe o morfismo

$$\langle f, g \rangle : \Gamma_1 \cup \Gamma_2 \rightarrow \varphi \wedge \phi.$$

Pela regra de derivação (ax) é possível derivar $\varphi \wedge \phi \vdash \varphi \wedge \phi$, assim define-se aplicando $\wedge_{E_1}$ e $\wedge_{E_2}$, respectivamente, $\pi_1 : \varphi \wedge \phi \rightarrow \varphi$ e $\pi_2 : \varphi \wedge \phi \rightarrow \phi$.

Seja $eval_{\phi \rightarrow \varphi} : (\phi \rightarrow \varphi) \wedge \phi \vdash \varphi$ o morfismo associado com a prova D abaixo:

$$\dfrac{\dfrac{(\phi \rightarrow \varphi) \wedge \phi \vdash (\phi \rightarrow \varphi) \wedge \phi}{(\phi \rightarrow \varphi) \wedge \phi \vdash \phi \rightarrow \varphi}(\wedge_{E_1}) \qquad \dfrac{(\phi \rightarrow \varphi) \wedge \phi \vdash (\phi \rightarrow \varphi) \wedge \phi}{(\phi \rightarrow \varphi) \wedge \phi \vdash \phi}(\wedge_{E_2})}{(\phi \rightarrow \varphi) \wedge \phi \vdash \varphi}(\rightarrow_E)$$

Por fim, suponha o morfismo $g : \Gamma \wedge \phi \rightarrow \varphi$, que está associado à prova $D \equiv \Gamma \wedge \phi \vdash \varphi$. Então, ao aplicar a regra $(\rightarrow_I)$ em D tem-se prova $\Gamma \vdash \phi \rightarrow \varphi$, que está associada ao morfismo $\Lambda_g : \Gamma\phi \rightarrow \varphi$.

Teorema 5.3. A categoria **CND** é uma categoria cartesiana fechada.

(i) O objeto terminal é a proposição $\top$, pois para qualquer proposição $\phi \in \bar{P}$ existe uma única seta $\bigcirc_\phi : \phi \rightarrow \top$.

(ii) O produto é a tripla $(\phi \wedge \varphi, \pi_1 : \phi \wedge \varphi \rightarrow \phi, \pi_2 : \phi \wedge \varphi \rightarrow \varphi)$. Pois, para qualquer objeto $\Gamma \in Obj_{CND}$ e todo par de morfismos $f : \Gamma \rightarrow \phi$ e $g : \Gamma \rightarrow \varphi$, pela regra $\rightarrow_I$ existe um morfismo $\langle \phi \wedge \varphi \rangle : \Gamma \rightarrow \phi \wedge \varphi$ que é único e faz o seguinte diagrama comutar:

$$\begin{tikzcd}
 & \Gamma \arrow[dl, "f"'] \arrow[d, dashed, "\langle \phi \wedge \varphi \rangle"] \arrow[dr, "g"] & \\
\phi & \phi \wedge \varphi \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & \varphi
\end{tikzcd}$$

Figura 5.1: Diagrama comutativo do produto entre ϕ e φ.

(iii) O objeto exponencial é o par $(\phi \to \varphi, eval_{\phi\to\varphi} : (\phi \to \varphi) \wedge \phi \to \varphi)$. Suponha $g : \Gamma \wedge \phi \to \varphi$, então pela regra $(\to_I)$ existe um outro único morfismo $\Lambda_g : \Gamma \to (\phi \to \varphi)$. Assim, é possível construir o morfismo $\Lambda_g \times id_\phi = \langle \Lambda_g \circ \pi_1, id_\phi \circ \pi_2 \rangle : \Gamma \wedge \phi \to (\phi \to \varphi) \wedge \phi$, sendo $\pi_1 : \phi \wedge \varphi \to \phi$ e $\pi_2 : \phi \wedge \varphi \to \varphi$, que é o único que faz o seguinte diagrama comutar:

$$\begin{tikzcd}
\Gamma \arrow[d, dashed, "\Lambda_g"] & \Gamma \wedge \phi \arrow[d, "\Lambda_g \times Id_\phi"'] \arrow[dr, "g"] & \\
\phi \to \varphi & (\phi \to \varphi) \wedge \phi \arrow[r, "eval_{\phi\to\varphi}"'] & \varphi
\end{tikzcd}$$

Figura 5.2: Diagrama comutativo do objeto exponencial na categoria **CND**.

O diagrama da Figura 5.2 é basicamente sobre a equivalência lógica

$$\phi \wedge \varphi \to \tau \equiv \phi \to (\varphi \to \tau),$$

que é generalizada pela Definição 4.17 de Objeto Exponencial e pela equivalência de coleções

$$Hom(C \times A, B) \leftrightarrow Hom(C, B^A).$$

Como já registrado nesta dissertação, a função $\Lambda_g : \Gamma \to (\phi \to \varphi)$ é conhecida como *currying* e permite simular o morfismo $g : \Gamma \wedge \phi \to \varphi$ parcialmente aplicado em Γ.

5.3.2 Cálculo Lambda Tipado é uma Categoria Cartesiana Fechada

Segundo (PIERCE, 1991) uma das correspondências mais citadas na Ciência da Computação é a relação entre Cálculo Lambda Tipado e Categorias Cartesianas Fechadas, visto que uma das principais aplicações da Teoria das Categorias na Ciência da Computação é na descrição semântica de linguagens de programação.

A demostração da interpretação categórica do Cálculo Lambda Tipado (sistema TA) para uma categoria cartesiana fechada **CTA** é, em resumo, interpretar os tipos como objetos e funções como morfismos. Cria-se um conjunto parcialmente ordenado que defini uma categoria **CTA** que é cartesiana fechada.

A conversão de um contexto para um tipo é realizada por uma função $[\![]\!]$, tal que

$$[\![\emptyset]\!] = 1;$$
$$[\![\Gamma \cup \{x : \alpha\}]\!] = [\![\Gamma]\!] \times \alpha,$$

onde α é um tipo paramétricos qualquer.

Nota 5.6. Por simplicidade, a notação $[\![A]\!]$ é abreviada para apenas A.

O restante da prova é similar à anterior. Primeiro é criado a relação de pré-ordem entre uma interpretação de um contexto $[\![A]\!] = \tau$ e um tipo α de um termo M, tal que

$$\tau \leq \alpha \Longleftrightarrow \text{existe uma prova de } A \vdash M : \alpha$$

A reflexividade da relação é provada pela regra (var). A transitividade é garantida pela substituição em TA e pelo Lema 5.1 da Substituição de provas do sistema ND[1]. Toma-se então $A \leq \tau$ e $\tau \leq \alpha$ a partir, respectivamente, de

$$\frac{\Sigma}{A \vdash M : \tau} \text{ e } \frac{\Pi}{A \cup \{x : \tau\} \vdash N : \alpha} \text{ . Então } \frac{\mathbb{S}(\Pi, \Sigma)}{A \vdash [M/x]N : \alpha},$$

logo $A \leq \alpha$. A antisimetria tem o mesmo problema e solução que na prova anterior. Portanto, é criado a seguinte relação de equivalência entre os tipos $\alpha \equiv [\![A]\!]$ e $\tau \equiv [\![B]\!]$:

$$\tau \simeq \alpha \Longleftrightarrow \text{ se há provas de } A \vdash M : \tau \text{ e } B \vdash N : \alpha.$$

A classe de equivalência abaixo é definida a partir do conjunto $\mathbb{T}_p$ de tipos de TA.

$$[\tau] = \{\alpha \in \mathbb{T}_p \mid \alpha \simeq \tau\}.$$

E, por fim, o conjunto $\bar{T}$ definido por

$$\bar{T} = \{[\tau] \mid \tau \in P\}.$$

Definição 5.5. Categoria **CTA**

(a) Obj_{CTA} é o conjunto $\bar{T}$, ou seja, os objetos são as classes de equivalência.

[1] O Isomorfismo de Curry-Howard permite utilizar teoremas e lemas do sistema ND no sistema TA.

(b) Mor_{CTA} é definido pela relação $\leq$. Assim, para quaisquer $\tau, \alpha \in \bar{T}$ existe uma seta/morfismo $f : \tau \to \alpha$ se, e somente se, a relação $\tau \leq \alpha$ é verdadeira.

Como morfismos são gerados por provas, então são obtidos pelas seguintes regras do sistema TA adaptadas:

$$\frac{}{f : A \times \tau \to \tau}\,(var) \qquad \frac{}{\bigcirc_A : A \to 1}\,(unit)$$

$$\frac{f : A_1 \to \alpha \to \tau \qquad g : A_2 \to \alpha}{eval_{\tau\to\alpha} \circ \langle f, g\rangle : A_1 \times A_2 \to \tau}\,(app) \qquad \frac{g : A \times \alpha \to \tau}{\Lambda_g : A \to (\alpha \to \tau)}\,(abs)$$

$$\frac{f : A_1 \to \alpha \qquad g : A_2 \to \tau}{\langle f, g\rangle : A_1 \times A_2 \to \alpha \times \tau}\,(pair)$$

$$\frac{f : A \to \alpha \times \tau}{\pi_1 \circ f : A \to \alpha}\,(\pi_1) \qquad \frac{f : A \to \alpha \times \tau}{\pi_2 \circ f : A \to \tau}\,(\pi_2)$$

(c) ∂_0 e ∂_1 são as funções que levam, respectivamente, uma relação à sua classe de equivalência de origem e destino.

(d) $\circ$ é a composição da relação $\leq$ e é provada pela transitividade da mesma. Portanto, se $f : A \leq \alpha$ e $g : \alpha \to \tau$, então $g \circ f : A \to \tau$.

(e) ι retorna a relação reflexiva $\alpha \vdash \alpha$ de cada classe de equivalência $[\alpha]$.

Teorema 5.4. A categoria **CTA** é uma categoria cartesiana fechada.

(i) O objeto terminal é o tipo 1, pois para qualquer proposição $\tau \in \bar{T}$ existe uma única seta $\bigcirc_\tau : \tau \to 1$.

(ii) O produto é a tripla $(\tau \times \alpha, \pi_1 : \tau \times \alpha \to \tau, \pi_2 : \tau \times \alpha \to \alpha)$. Pois, para qualquer objeto $A \in Obj_{CTA}$ e todo par de morfismos $f : A \to \tau$ e $g : A \to \alpha$, pela regra (abs) existe um morfismo $\langle \tau \times \alpha\rangle : A \to \tau \times \alpha$ que é único e faz o seguinte diagrama comutar:

$$\begin{tikzcd}
 & A \arrow[dl, "f"'] \arrow[d, dashed, "\langle \tau \times \alpha\rangle"] \arrow[dr, "g"] & \\
\tau & \tau \times \alpha \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & \alpha
\end{tikzcd}$$

Figura 5.3: Diagrama comutativo do produto entre τ e α.

(iii) O objeto exponencial é o par $(\tau \to \alpha, eval_{\tau\to\alpha} : (\tau \to \alpha) \times \tau \to \alpha)$. Suponha $g : A \times \tau \to \alpha$, então pela regra (abs) existe um outro único morfismo $\Lambda_g : A \to (\tau \to \alpha)$. Assim, é possível construir o morfismo $\Lambda_g \times id_\tau = \langle \Lambda_g \circ \pi_1, id_\tau \circ \pi_2 \rangle : A \times \tau \to (\tau \to \alpha) \times \tau$, sendo $\pi_1 : \tau \times \alpha \to \tau$ e $\pi_2 : \tau \times \alpha \to \alpha$, que é o único que faz o seguinte diagrama comutar:

$$\begin{tikzcd}
A \arrow[d, dashed, "\Lambda_g"] & A \times \tau \arrow[d, "\Lambda_g \times Id_\tau"'] \arrow[dr, "g"] & \\
\tau \to \alpha & (\tau \to \alpha) \times \tau \arrow[r, "eval_{\tau\to\alpha}"'] & \alpha
\end{tikzcd}$$

Figura 5.4: Diagrama comutativo do objeto exponencial na categoria **CTA**.

6 Análise do Sistema de Tipos de Haskell

Como já mencionado neste trabalho, Haskell é uma linguagem de programação funcional pura e que utiliza diversos conceitos de Teoria dos Tipos e Teoria das Categorias. A análise desta seção foca em ser matematicamente coerente quanto às alegações geralmente feitas sobre o sistema de tipos de Haskell, principalmente do ponto de vista da lógica intuicionista e da Teoria das Categorias. Assim, busca-se identificar características positivas e falhas na fundamentação teórica do sistema de tipos de Haskell.

6.1 Visão Lógica do Sistema de Tipos de Haskell

Curry-Howard relaciona sistemas de tipos com o fragmento intuicionista da Dedução Natural de Gentzen. Cabe agora verificar como isso influencia o sistema de tipos de Haskell.

A visão intuicionista da implicação lógica afirma que $A \to B$ é um procedimento que produz uma construção de B se fornecida uma construção de A. Em Haskell, como exemplificado em (WIKIBOOKS...,), 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 verdadeiras e o retorno como conclusão.

Em Haskell existe o termo `undefined :: a` , que serve para representar uma computação com erro ou um loop infinito, e sua avaliação gera uma exceção como abaixo.

```
Prelude> undefined
*** Exception: Prelude.undefined
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/Err.hs:79:14 in base:GHC.Err
  undefined, called at <interactive>:1:1 in interactive:Ghci1
```

Uma possível implementação do `undefined` é 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[1]. Apesar de não existir especificação formal para semântica operacional de Haskell, qualquer não terminação é considerada como um valor *bottom* ($\perp$).

Uma vez que existem termos com o tipo `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.

No último capítulo foi mostrado que o Isomorfismo de Curry-Howard além de relacionar tipos com proposições e termos com provas, também relaciona tipos habitados com teoremas. De acordo com (WIKIBOOKS...,), 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-a concluir que o sistema de tipos de Haskell é inconsistente.

No entanto, afirmar que Haskell é inconsistente pode ser uma imprecisão, pois segundo (COSTA; ABE, 2000):

> 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 ($\perp$)[2]. Portanto, seria mais

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

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

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
```
[1],

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 sistema de tipos de Haskell garante 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 composição de funções, que poderiam resultar em um programa sem sentido semântico. Essa garantia é conhecida como *Type Safety*, que é conceituada por (PIERCE, 2002) como:

> A propriedade mais básica deste tipo de sistema ou qualquer outro é a segurança (*type safety*, também chamada de *soundness*): termos bem tipados não "dão errado". Nós já escolhemos como formalizar o que significa para um termo dar errado: significa alcançar um "estado *stuck*"[2] que não é designado como valor final, mas onde as regras de redução não nos dizem o que fazer em seguida. O que queremos saber, então, é que termos bem tipados não ficam *stuck*.

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

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

Que captura essencialmente a ideia do erro somar uma `String` com um `Int` (por exemplo o termo `"1" + 1`), pois resultaria num termo sem redução possível (*stuck*). Mais precisamente, *Type Safety* = *Progress* + *Preservation*, sendo

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

(ii) *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 (DAMAS; MILNER, 1982) 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.

6.1.1 *Theorems for free* em Haskell

Theorems for free! (ou Teoremas de graça!) é o nome de um artigo de Philip Wadler (WADLER; BLOTT, 1989), 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 (REYNOLDS, 1983). *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 seguinte igualdade:

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

que pode ser reescrito, 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 $\perp$, 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.

Nota 6.1. Haskell é uma linguagem com avaliação preguiçosa, ou seja, somente tenta avaliar uma expressão para forma normal (sem redex) quando o topo da sua árvore sintática é avaliado. Consequentemente, deve avaliar os seus ramos para forma normal de cabeça, ou *weak head normal form* (WHNF), e para forma normal somente se necessário (ex: tornou-se o topo de uma árvore sintática que está sendo avaliado). Um expressão está em WHNF se o termo mais no topo da sua árvore sintática for:

- Um construtor de valor (ex: `1` , `Just 1` , `[undefined]`).
- Uma abstração lambda (ex: `\x -> x` , `\x -> 1 + 1`).
- Uma função aplicada parcialmente (ex: `(+1)` , `:[]`).

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 básicas 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` apenas com Cálculo Lambda, 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 $\perp$. Então, tenta-se a redução do primeiro argumento. Se o primeiro

argumento tem *weak head normal form* 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]) = [].
```

6.2 Visão Categórica do Sistema de Tipos de Haskell

Como visto no capítulo anterior, um Cálculo Lambda Tipado projeta uma categoria cartesiana fechada, portanto respeita regras básicas, como: a identidade para qualquer objeto e a composição de morfismos é associativa. O fato de ser cartesiana fechada implica na existência de objeto terminal, todos os produtos e 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.

A página *Wiki* do Haskell (HASK...,) 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**.

Como apontado por Andrej Bauer em seu blog (MATHEMATICS...,), 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 algum desses trabalhos 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. Como afirmado pela *Wiki* do Haskell, considerar funções totais permite 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.

As demais funções parciais são identificáveis, basta verificar se os argumentos tratam todos os valores possíveis através de variáveis e se os valores dos casamentos de padrão capturaram cada um dos valores do tipo do argumento. Assim, essas funções parciais (que não são de recursão geral) podem ser representadas por meio do dado algébrico `Maybe`, por exemplo:

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

Definição 6.1. Categoria **PHask**

(a) 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 () = ()`).

(b) 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.

(c) ∂_0 e ∂_1 são as funções que levam, respectivamente, uma função ao seu tipo de origem e destino.

(d) A composição é dada pela função `(.) :: (b -> c) -> (a -> b) -> a -> c` e a prova da associatividade é a mesma utilizada em **Set**.

(e) 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 6.1. A categoria **PHask** é uma categoria cartesiana fechada.

1. 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 `()`. Exemplos de funções que geram o morfismo

`c : Int` $\rightarrow$ `()` são `constInt _ = ()`, `constInt2 x = ()` e `constInt . id`, pois têm o mesmo tipo e mapeamento. Além disso, o objeto terminal também resgata a noção de elemento na categoria **PHask**: um valor constante do tipo `Int` (ou uma função nulária para este valor) gera o morfismo `()` $\rightarrow$ `Int`, pois o valor `1 :: Int` é único em relação a função `const_1 () = 1 :: () -> Int`.

2. 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.

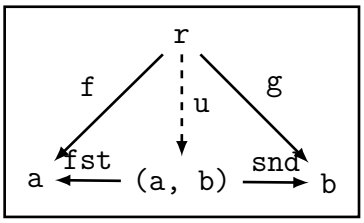

Figura 6.1: Diagrama comutativo do produto em Haskell.

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

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

3. O objeto exponencial em Haskell é formado pelo objeto `b -> c`[1] 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 abaixo comutar.

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

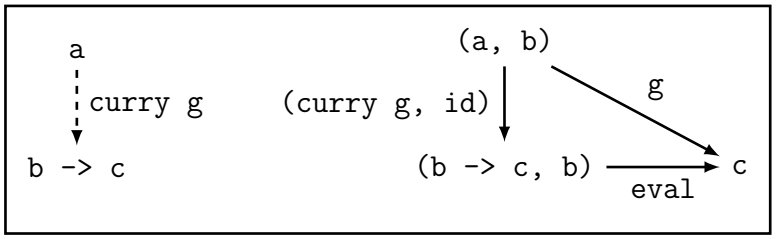

Figura 6.2: Diagrama comutativo do objeto exponencial em Haskell.

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 implicito 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 $\mathtt{m}_1$ `: a` $\rightarrow$ `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 $\mathtt{m}_2$ `: (a, b)` $\rightarrow$ `a`, logo `(a, b)` $\rightarrow$ `a` $\equiv$ `a` $\rightarrow$ `b -> a`.

6.2.1 Visão Categórica de Classes de Tipos

As classes de tipos do Haskell, descritas no Capítulo 3, 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 monóide e funtor, e verifica se são as estruturas que alegam ser.

Um monóide 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 $\boldsymbol{e}$ 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 `Either` recebe dois tipos `a, b` e retorna `Either a b`. Uma função cujo mapeamento é entre tipos não tem um tipo, na realidade tem algo chamado de *kind*. O *kind* de `Either` 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, 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.

7 Conclusões

Este trabalho apresentou os fundamentos teóricos da pesquisa em sistemas de tipos: Teoria dos Tipos e Teoria das Categorias, com a finalidade de servir como base para a tripla relação Curry-Howard-Lambek e como fundamentação da análise categórica do sistema de tipos de Haskell.

Pode-se presumir o que é um sistema de tipos e como ele se relaciona com Cálculo Lambda Tipado por meio do estilo de Curry chamado *Type Assignment*. Ainda apresentou-se teoremas importantes como o de Church-Rosser e a Forte Normalização do Cáculo Lambda Tipado. Estes teoremas são fundamentais em sistemas de tipos de linguagens de programação, pois estão relacionados com a possibilidade de alcançar um resultado único para cada computação.

Alguns sistemas de tipos foram apresentados a fim de servir como base teórica para análise realizada no Capítulo 6 e como uma visão geral dos principais sistemas de tipos. Damas-Milner é um clássico sistema de tipos com um algoritmo de inferência decidível, por isso é utilizado em diversas linguagens de programação funcional. Já o Sistema-F é uma generalização do DM, mas não tem inferência decidível. O sistema AHDM é que mais se aproxima do sistema de tipos de Haskell, pois conta sobrecarga de operadores.

Da Teoria das Categorias salientou-se conceitos básicos como a definições de categoria e de diagrama, e alguns dos principais tipos de estruturas como funtores e limites. Um exemplo de categoria de uma linguagem de programação funcional foi criado, que permitiu compreender como a semântica de tipos e funções pode ser modelada em uma categoria, a qual, por fim, apresentou uma certa similaridade com a categoria **PHask**.

O sistema de tipos de Haskell não representa uma lógica consistente, pois devido ao termo `undefined :: a` 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 precisa ser *type safe*. O uso de *Theorems for free* em Haskell é possível, mas enfraquecido por causa do termo `undefined :: a` e da função

`seq :: a -> b -> b`, portanto mesmo que o sistema de tipos de Haskell não seja um lógica consistente, Curry-Howard ainda trás vantagens.

Curry-Howard-Lambek é utilizado no sistema de tipos de Haskell, não somente para definir algumas classes de tipos, mas também para garantir o bom comportamento do sistema. Porém, foram identificados alguns problemas, em particular a ausência de uma semântica operacional que seria fundamental para definir uma categoria dos tipos e funções. Como alternativa foi construído a categoria **PHask** a partir de um subconjunto de Haskell com apenas funções totais.

Os elementos categóricos de Haskell não (necessariamente) estão de acordo com a teoria: uma instância da classe `Monoid` não precisa respeitar as leis dos monoides. Assim, uma proposta de trabalho é estender o sistema AHDM de maneira que a assinatura de tipo garanta propriedades de morfismos, portanto as classes de tipos poderiam refletir corretamente elementos categóricas.

Referências Bibliográficas

ABRAMSKY, S.; TZEVELEKOS, N. Introduction to categories and categorical logic. In: *New structures for physics*. [S.l.]: Springer, 2010. p. 3–94.

ASPERTI, A.; LONGO, G. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist*. Cambridge, MA, USA: MIT Press, 1991. ISBN 0-262-01125-5.

BARR, M.; WELLS, C. (Ed.). *Category Theory for Computing Science, 2Nd Ed.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1995. ISBN 0-13-323809-1.

BELL, J. L. Types, sets and categories. *Akihiro Kanamory Handbook of the History of Logic*, v. 6, 2012.

BELL, J. L. *The Development of Categorical Logic*. 2014.

BERGER, C. A categorical approach to proofs-as-programs. 2010.

CARDONE, F.; HINDLEY, J. R. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, v. 5, 2006.

CHURCH, A. A set of postulates for the foundation of logic part i. *Annals of Mathematics*, v. 33, n. 2, p. 346–366, 1932.

CHURCH, A. A set of postulates for the foundation of logic part ii. *Annals of Mathematics*, v. 34, n. 2, p. 839–864, 1933.

CHURCH, A.; ROSSER, J. B. Some properties of conversion. *Transactions of the American Mathematical Society*, v. 39, p. 472–482, 1936.

COSTA, N. C. d.; ABE, J. M. Paraconsistência em informática e inteligência artificial. *Estudos Avançados*, scielo, v. 14, p. 161 – 174, 08 2000. ISSN 0103-4014.

CURRY, H.; FEYS, R. *Combinatory Logic*. [S.l.]: North-Holland Publishing Company, 1958. (Combinatory Logic, v. 1).

CURRY, H. B. Modified basic functionality in combinatory logic. *Dialectica*, Blackwell Publishing Ltd, v. 23, n. 2, p. 83–92, 1969. ISSN 1746-8361.

DAMAS, L. *Type assignment in programming languages*. Tese (Doutorado), 1985. PHD.

DAMAS, L.; MILNER, R. Principal type-schemes for functional programs. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1982. (POPL '82), p. 207–212. ISBN 0-89791-065-6.

EILENBERG, S.; LANE, M. General theory of natural equivalences. *Trans. Amer. Math. Soc.*, v. 58, p. 231–294, 1945.

FERNANDES, F. L. *O Isomorfismo de Curry-Howard via Teoria de Categorias*. Belo Horizonte, Brasil: [s.n.], jul 2011. Monografia (Bacharel em Matemática).

GENTZEN, G. Investigations into Logical Deduction. In: SZABO, M. E. (Ed.). *The Collected Papers of Gerhard Gentzen*. Amsterdam: North-Holland, 1969. p. 68–213.

HANKIN, C. *An Introduction to Lambda Calculi for Computer Scientists*. [S.l.]: Kings College, 2004. (Texts in computing). ISBN 9780954300654.

HASK - HaskellWiki. Acessado: 23/05/2017. Disponível em: <https://wiki.haskell.org/Hask>.

HINDLEY, J. *Basic Simple Type Theory*. [S.l.]: Cambridge University Press, 1997. (Cambridge Tracts in Theoretical Computer Science). ISBN 9780521465182.

HINDLEY, J. R.; SELDIN, J. P. *Lambda-Calculus and Combinators: An Introduction*. 2. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521898854, 9780521898850.

HINDLEY, R. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, American Mathematical Society, v. 146, p. 29–60, 1969. ISSN 00029947.

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*. [S.l.]: Academic Press, 1980. p. 479–490.

HUDAK, P. et al. A history of haskell: Being lazy with class. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. New York, NY, USA: ACM, 2007. (HOPL III), p. 12–1–12–55. ISBN 978-1-59593-766-7.

JONES, S. P. *Haskell 98 language and libraries : the revised report*. [S.l.]: Cambridge University Press, 2003. Hardcover. ISBN 0521826144.

KALMAN, J. A. Condensed detachment as a rule of inference. *Studia Logica*, v. 42, n. 4, p. 443–451, 1983. ISSN 1572-8730.

LAMBEK, J. Cartesian closed categories and typed lambda- calculi. In: *Proceedings of the Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*. London, UK, UK: Springer-Verlag, 1986. p. 136–175. ISBN 3-540-17184-3.

LAMBEK, J.; SCOTT, P. *Introduction to Higher-Order Categorical Logic*. [S.l.]: Cambridge University Press, 1988. (Cambridge Studies in Advanced Mathematics). ISBN 9780521356534.

LAMBEK, J.; SCOTT, P. J. *Introduction to Higher Order Categorical Logic*. New York, NY, USA: Cambridge University Press, 1986. ISBN 0-521-24665-2.

MATHEMATICS and Computation - Hask is not a category. Acessado: 25/05/2017. Disponível em: <http://math.andrej.com/2016/08/06/hask-is-not-a-category>.

MENEZES, P.; HAEUSLER, E. H. *Teoria das Categorias para Ciência da Computação*. first. [S.l.]: Editora Sagra Luzzatto, 2001. (Livros Didáticos, v. 12).

MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, v. 17, p. 348–375, 1978.

MITCHELL, J. C. *Foundations of Programming Languages*. Cambridge, MA, USA: MIT Press, 1996. ISBN 0-262-13321-0.

MOGGI, E. Notions of computation and monads. *Inf. Comput.*, Academic Press, Inc., Duluth, MN, USA, v. 93, n. 1, p. 55–92, jul. 1991. ISSN 0890-5401.

PIERCE, B. C. *Basic Category Theory for Computer Scientists*. Cambridge, MA, USA: MIT Press, 1991. ISBN 0-262-66071-7.

PIERCE, B. C. *Types and Programming Languages*. 1st. ed. [S.l.]: The MIT Press, 2002. ISBN 0262162091, 9780262162098.

REYNOLDS, J. C. Types, abstraction and parametric polymorphism. In: *IFIP Congress*. [S.l.: s.n.], 1983. p. 513–523.

RYDEHEARD, D. E.; BURSTALL, R. M. *Computational Category Theory*. [S.l.]: Prentice Hall, 1988.

SCOTT, D. S. Relating theories of the lambda calculus. *To HB Curry: Essays on combinatory logic, lambda calculus and formalism*, Academic Press New York, p. 403–450, 1980.

SEBESTA, R. W. *Concepts of Programming Languages*. 10th. ed. [S.l.]: Pearson, 2012. ISBN 0273769103, 9780273769101.

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

TAKAHASHI, M. Parallel reductions in λ-calculus. *Journal of Symbolic Computation*, v. 7, n. 2, p. 113 – 123, 1989. ISSN 0747-7171.

TATE, R.; STEPP, M.; LERNER, S. Generating compiler optimization from proofs. In: . [S.l.: s.n.], 2012.

WADLER, P. Monads for functional programming. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995. p. 24–52. ISBN 3-540-59451-5.

WADLER, P. Propositions as types. *Commun. ACM*, ACM, New York, NY, USA, v. 58, n. 12, p. 75–84, nov. 2015. ISSN 0001-0782.

WADLER, P.; BLOTT, S. How to make ad-hoc polymorphism less ad hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1989. (POPL '89), p. 60–76. ISBN 0-89791-294-2.

WELLS, J. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, v. 98, n. 1, p. 111 – 156, 1999. ISSN 0168-0072.

WIKIBOOKS: Haskell/The Curry-Howard isomorphism. Acessado: 18/05/2017. Disponível em: <https://en.wikibooks.org/wiki/Haskell/TheCurry-Howardisomorphism>.

Apêndice

A Algoritmo de inferência do sistema TA

```
type Infer = ExceptT TErro (S.State Count)

infer :: Expr -> Context -> Infer (Type, Context)
infer e c = case e of
    Var e' -> do
            (te, ce') <- lookContext e' c
            return (te, ce')

    (Abs x e') -> do
            (te', c') <- infer e' c
            (tx, c'') <- lookContext x c'
            let c''' = Map.delete x c''
            let ta = TArrow tx te'
            return (ta, c''')

    (App e' e'') -> do
        (te', c') <- infer e' c
        (te'', c'') <- infer e'' c
        case te' of
            TArrow r _ -> do
                ntv <- newTVar
                let lc' = (Map.elems (Map.intersection c' c'')) ++ [leftType te', ntv]
                let lc'' = (Map.elems (Map.intersection c'' c')) ++ [te'', ntv]
                s <- unifyE lc' lc''
                let ste' = tsubst te' s
                let rste' = rightType ste'
                let cu = Map.union c' c''
                let c''' = Map.map (flip tsubst s) cu
                return (rste', c''')
            TVar _ -> do
                ntv <- newTVar
                ntv' <- newTVar
                let lc' = (Map.elems (Map.intersection c' c'')) ++ [te', ntv]
                let lc'' = (Map.elems (Map.intersection c'' c')) ++ [TArrow te'' ntv', ntv]
                s <- unifyE lc' lc''
                let sntv' = tsubst ntv' s
                let cu = Map.union c' c''
                let c''' = Map.map (flip tsubst s) cu
                return (sntv', c''')
```