UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

Maycon Jose Jorge Amaro
Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro

SEMÂNTICA DENOTACIONAL PARA O LAMBDA CÁLCULO COMPUTACIONAL EM AGDA

Ouro Preto, MG
2020

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

Maycon Jose Jorge Amaro

SEMÂNTICA DENOTACIONAL PARA O LAMBDA CÁLCULO COMPUTACIONAL EM AGDA

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro

Ouro Preto, MG
2020

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

A485s Amaro, Maycon José Jorge.
Semântica denotacional para o lambda cálculo computacional em Agda. [manuscrito] / Maycon José Jorge Amaro. - 2020.
26 f.

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Graduação em Ciência da Computação .

1. Linguagem de programação (Computadores) - Semântica. 2. Linguagem de programação (Computadores) - Lambda cálculo. 3. Conjuntos - Teoria dos tipos. I. Ribeiro, Rodrigo Geraldo. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.43:510.27

Bibliotecário(a) Responsável: Sione Galvão Rodrigues - CRB6 / 2526

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

MAYCON JOSÉ JORGE AMARO

Semântica denotacional para o lambda-cálculo computacional em Agda

Membros da banca:
Rodrigo Geraldo Ribeiro (orientador) - Doutor - Universidade Federal de Ouro Preto (UFOP)
Dayanne Gouveia Coelho (examinadora) - Doutora - Universidade Federal de Ouro Preto (UFOP)
Samuel da Silva Feitosa (examinador) - Doutor - Instituto Federal de Santa Catarina (IFSC)

Versão final aprovada em 05 de Novembro de 2020.

De acordo,
Presidente e Prof. Orientador Rodrigo Geraldo Ribeiro

Documento assinado eletronicamente por **Rodrigo Geraldo Ribeiro**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 10/11/2020, às 13:39, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0100736** e o código CRC **C635AD73**.

Referência: Caso responda este documento, indicar expressamente o Processo nº 23109.008502/2020-28 SEI nº 0100736

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Resumo

Novos recursos para linguagens de programação precisam de uma base teórica sólida antes de serem implementados, e frequentemente são modelados através de um sistema menor—os λ-cálculos—capaz de capturar a essência das linguagens e separá-las dos *syntatic sugars*. A semântica denotacional desses sistemas é extremamente útil para o racícionio sobre programas e suas propriedades. Este trabalho formaliza, em Agda, o modelo padrão de uma semântica denotacional para o λ-cálculo computacional, que estende o λ-cálculo tipado simples com um operador modal usado para definir computações envolvendo diferentes efeitos colaterais.

Palavras-chave: semântica formal, lambda cálculo, teoria de tipos.

Abstract

New features for programming languages require solid foundations before implementation. These features are often modeled by smaller systems—the λ-calculi—that capture the language's core and separate them from the syntatic sugars. The denotational semantics of those systems are a very helpful tool in reasoning about programs and their properties. This work formalizes, in Agda, the standard model for the computational λ-calculus, a system that extends the simply typed λ-calculus with a modal operator used to define computations involving different side effects.

Keywords: formal semantics, lambda calculus, type theory.

Lista de Abreviaturas e Siglas

CLC	Computational Lambda Calculus
NbE	Normalization by Evaluation
STLC	Simply Typed Lambda Calculus

Lista de Símbolos

α	Letra grega alfa
β	Letra grega beta
η	Letra grega eta
γ	Letra grega gamma
Γ	Letra grega grande gamma
λ	Letra grega lambda
μ	Letra grega mu
τ	Letra grega tau
$\varnothing$	Conjunto vazio
$\rightarrow$	Construtor de tipos funcionais
$\longrightarrow$	Avaliação entre expressões
$\Downarrow$	Avaliação entre expressão e valor final
$\Leftarrow$	Atribuição
∇	Atribuidor de tipo com base no contexto
$\top$	Verdadeiro
$\bot$	Falso
$\neg$	Operador de negação lógica
$\vdash$	Sequente
$*$	Coringa para qualquer tipo

Sumário

1 Introdução

Na Ciência da Computação, em particular no estudo das linguagens de programação, definir o significado de símbolos e sentenças de maneira rigorosa é um processo fundamental para a sua implementação. Algumas abordagens de definição desses significados incluem Operacional e Denotacional, cada uma com suas aplicações mais adequadas. Normalização—padronizar algum objeto, geralmente com a intenção de reduzir alguma redundância—é um exemplo de aplicação que faz bom uso da abordagem denotacional. Um conhecido sistema em que há um grande interesse na definição dessas semânticas é o λ-cálculo, que tem sido amplamente utilizado na especificação de recursos de linguagens de programação e no estudo de sistemas de tipos (PIERCE, 2002). Proposto por Moggi (1988), o λ-cálculo computacional é extensão do λ-cálculo tipado simples que permite definir computações contendo efeitos colaterais, servindo assim, como um modelo para análise dessa classe de programas.

Recentemente, o uso de assistentes de provas como Coq e Agda tem-se tornado frequente na literatura sobre métodos formais. Tais assistentes exigem uma descrição minuciosa de demonstrações e as submetem a uma verificação automática. Neste trabalho, pretende-se formalizar uma semântica denotacional para o λ-cálculo computacional usando o assistente de provas Agda.

1.1 Justificativa

Projetistas de linguagens de programação tem explorado novos recursos e técnicas para as linguagens modernas, para ao mesmo tempo permitir mais expressividade sem necessariamente sacrificar desempenho ou segurança. Esses recursos precisam de uma base teórica sólida para serem implementados. A formalização proposta nesse trabalho é uma contribuição nesse caminho, não de ser uma aplicação imediata, mas de contribuir com a boa fundamentação da teoria subjacente a recentes desenvolvimentos no uso de tipos para controle de efeitos em programas.

1.2 Objetivos

O objetivo principal deste trabalho é a formalização da semântica denotacional para o λ-cálculo computacional usando a linguagem Agda.

1.3 Organização do Trabalho

No Capítulo 2, apresenta-se os conceitos prévios e trabalhos relacionados; no Capítulo 3 apresenta-se um pouco sobre o assistente de provas e linguagem Agda; no Capítulo 4 apresenta-se

os principais sistemas de λ-cálculo; no Capítulo 5 é apresentada e formalizada a semântica denotacional e no Capítulo 6 são apresentadas as considerações finais. Todo o código pode ser encontrado no seguinte repositório do GitHub: <https://github.com/mayconamaro/clc-denotational-agda>.

2 Revisão Bibliográfica

O objetivo deste capítulo é preparar o leitor para o tema abordado neste trabalho: formalização da semântica denotacional do λ-cálculo computacional. Para isso, apresenta-se as noções elementares de semântica formal. O λ-cálculo será apresentado em um capítulo dedicado. Ao final deste capítulo, apresenta-se alguns trabalhos relacionados.

2.1 Fundamentação Teórica

Para acompanhar melhor as definições da Subseção 2.1.1, considere a linguagem de exemplo abaixo, definida como uma gramática livre de contexto. Note que não há significado prévio para nenhuma das construções sintáticas.

$$
\begin{array}{l}
e \to v \mid \mathsf{suc}\ e \mid \mathsf{case}\ e\ e\ e \mid \mathsf{isz}\ e \mid \mathsf{not}\ e \\
v \to z \mid f \mid t
\end{array}
$$

2.1.1 Semântica Formal

Semântica, num sentido amplo, é o estudo de significados de sentenças ou símbolos. Uma semântica para uma linguagem de programação é a definição do significado e/ou comportamento dos programas válidos, *i.e.* sintaticamente corretos, da linguagem. As três principais abordagens para se definir semânticas de linguagens de programação são Operacional, Denotacional e Axiomática (NIELSON; NIELSON, 2007).

2.1.1.1 Semântica Operacional

A Semântica Operacional define o comportamento dos programas especificando regras de como eles seriam executados numa máquina abstrata, idealizando um interpretador (MILEWSKI, 2018). Nessa abordagem, há interesse em como o efeito da computação é produzido. Se divide em Semântica Operacional Estrutural, popularizada por Plotkin (2004), e Semântica Natural, popularizada por Kahn (1987). A primeira detalha os passos individuais enquanto a segunda apenas relaciona os estados inicial e final da computação. A semântica operacional é a abordagem mais adequada quando a intenção é implementar a linguagem (NIELSON; NIELSON, 2007).

A semântica natural—também chamada de passo-grande—da linguagem de exemplo é definida a seguir. A notação $e \Downarrow v$ indica que a expressão e avalia para o valor final v.

$$\frac{e_1 \Downarrow z}{\text{isz } e_1 \Downarrow t}\ \text{(isz\_z)} \qquad \frac{}{v_1 \Downarrow v_1}\ \text{(id)} \qquad \frac{e_1 \Downarrow t \qquad e_2 \Downarrow v_2}{\text{case } e_1\ e_2\ e_3 \Downarrow v_2}\ \text{(case\_t)}$$

$$\frac{e_1 \Downarrow \text{suc } n_1}{\text{isz } e_1 \Downarrow f}\ \text{(isz\_suc)} \qquad \frac{e_1 \Downarrow f \qquad e_3 \Downarrow v_3}{\text{case } e_1\ e_2\ e_3 \Downarrow v_3}\ \text{(case\_f)}$$

$$\frac{e_1 \Downarrow t}{\text{not } e_1 \Downarrow f}\ \text{(not\_t)} \qquad \frac{e_1 \Downarrow f}{\text{not } e_1 \Downarrow t}\ \text{(not\_f)} \qquad \frac{e_1 \Downarrow n_1}{\text{suc } e_1 \Downarrow \text{suc } n_1}\ \text{(suc\_cong)}$$

Considerando que z significa 0, e t, f, respectivamente, significam verdadeiro e falso, a regra isz_z avalia para verdadeiro quando a expressão é o valor 0, enquanto a regra isz_suc avalia para falso para números que são sucessores de outros. As regras not_f e not_v invertem o valor lógico do verdadeiro e falso. A regra suc_cong define a congruência da construção de sucessor, isto é, o suc de uma expressão é igual ao suc da avaliação dessa expressão. As regras case_t e case_f definem as condicionais, avaliando para a segunda ou terceira expressão dependendo do valor lógico da primeira. A regra id é a identidade.

A semântica operacional estrutural, também chamada de passo-pequeno, é definida abaixo para a linguagem de exemplo. A notação $e_1 \longrightarrow e_2$ indica que a expressão e_1 é avaliada para a expressão e_2.

$$\frac{}{\text{isz } z \longrightarrow t}\ \text{(isz\_t)} \qquad \frac{e_1 \longrightarrow e_2}{\text{isz } e_1 \longrightarrow \text{isz } e_2}\ \text{(isz\_step)}$$

$$\frac{}{\text{not } f \longrightarrow t}\ \text{(not\_f)} \qquad \frac{}{\text{isz suc } n_1 \longrightarrow f}\ \text{(isz\_f)}$$

$$\frac{e_1 \longrightarrow e_2}{\text{not } e_1 \longrightarrow \text{not } e_2}\ \text{(not\_step)} \qquad \frac{e_1 \longrightarrow e_1'}{\text{case } e_1\ e_2\ e_3 \longrightarrow \text{case } e_1'\ e_2\ e_3}\ \text{(case\_step)}$$

$$\frac{}{\text{not } t \longrightarrow f}\ \text{(not\_t)} \qquad \frac{}{\text{case } f\ e_2\ e_3 \longrightarrow e_3}\ \text{(case\_f)}$$

$$\frac{e_1 \longrightarrow e_2}{\text{suc } e_1 \longrightarrow \text{suc } e_2}\ \text{(suc\_step)} \qquad \frac{}{\text{case } t\ e_2\ e_3 \longrightarrow e_2}\ \text{(case\_t)}$$

As regras isz_t e isz_f verificam se uma expressão equivale a 0, avaliando, respectivamente, para verdadeiro e falso. Nessa abordagem de passo pequeno, algumas regras possibilitam que um simples passo de avaliação seja executado, que é o caso de isz_step, not_step, suc_step e

case_step. As regras not_t e not_f invertem o valor lógico de verdadeiro e falso, e as regras case_f e case_t representam as condicionais. Ambas case_f e case_t só podem ser aplicadas aos *valores finais* t e f, o que justifica a existência de case_step. O mesmo vale para as outras regras com step.

Para comparação, apresenta-se a redução da cadeia **case (not (isz z)) (suc (suc z)) (suc z)** para **suc z** nas duas semânticas apresentadas. Começando pela semântica natural:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{z \Downarrow z}\,(\text{id})}{\text{isz } z \Downarrow t}\,(\text{isz\_z})}{\text{not } (\text{isz } z) \Downarrow f}\,(\text{not\_t}) \quad \dfrac{\dfrac{}{z \Downarrow z}\,(\text{id})}{\text{suc } z \Downarrow \text{suc } z}\,(\text{suc\_cong})}{\text{case } (\text{not } (\text{isz } z))(\text{suc } (\text{suc } z))(\text{suc } z) \Downarrow \text{suc } z}\,(\text{case\_f})$$

e o correspondente na semântica operacional estrutural:

$$\dfrac{\dfrac{\dfrac{\text{case } (\text{not } (\text{isz } z))(\text{suc } (\text{suc } z))(\text{suc } z)}{\text{case } (\text{not } t)(\text{suc } (\text{suc } z))(\text{suc } z)}\,(\text{case\_step + isz\_t})}{\text{case } f(\text{suc } (\text{suc } z))(\text{suc } z)}\,(\text{case\_step + not\_t})}{\text{suc } z}\,(\text{case\_f})$$

2.1.1.2 Semântica Denotacional

A Semântica Denotacional associa as construções da linguagem para objetos matemáticos conhecidos, de forma que provar propriedades sobre programas corresponde a provar propriedades sobre tais objetos matemáticos. Nessa abordagem, há interesse apenas no efeito da computação e nos conceitos essenciais da linguagem, abstraindo-se os detalhes da execução (NIELSON; NIELSON, 2007). Vale ressaltar que a Semântica Denotacional também destaca imediatamente as construções impossíveis na linguagem, que são as mesmas impossibilidades[1] às quais estão sujeitos os objetos matemáticos associados (PIERCE, 2002).

Define-se a seguir a semântica denotacional da linguagem de exemplo, associando as construções sintáticas para objetos matemáticos. A notação $\mathcal{S}[\![e]\!]$ representa a aplicação da função semântica $\mathcal{S}$ em e. Note como z e `suc` são definidos para formar o conjunto $\mathbb{N}$ dos números naturais.

[1] Essas impossibilidades são programas sem semântica definida, como por exemplo, programas com erro de tipo.

$\mathcal{S}[\![z]\!] = 0$	$\mathcal{S}[\![f]\!] = \bot$
$\mathcal{S}[\![\text{suc } n]\!] = 1 + \mathcal{S}[\![n]\!]$	$\mathcal{S}[\![v]\!] = \top$
$\mathcal{S}[\![\text{case } e_1\ e_2\ e_3]\!] = \mathcal{S}[\![e_2]\!]$	se $\mathcal{S}[\![e_1]\!] = \top$
$\mathcal{S}[\![\text{case } e_1\ e_2\ e_3]\!] = \mathcal{S}[\![e_3]\!]$	se $\mathcal{S}[\![e_1]\!] = \bot$
$\mathcal{S}[\![\text{isz } e]\!] = \top$	se $\mathcal{S}[\![e]\!] = 0$
$\mathcal{S}[\![\text{isz } e]\!] = \bot$	se $\mathcal{S}[\![e]\!] > 0$
$\mathcal{S}[\![\text{not } e]\!] = \top$	se $\mathcal{S}[\![e]\!] = \bot$
$\mathcal{S}[\![\text{not } e]\!] = \bot$	se $\mathcal{S}[\![e]\!] = \top$

Nesta definição, a função semântica associa z ao número 0 e t, f aos valores lógicos $\top, \bot$, respectivamente. A construção suc é associada ao sucessor de um número. As construções not e isz são associadas à constantes dependendo do valor de seus argumentos, e case é associado à semântica de um de seus argumentos, dependendo da semântica do primeiro. Utilizando o mesmo exemplo para a semântica operacional, pode-se demonstrar, com esta semântica que a cadeia **case (not (isz z)) (suc (suc z)) (suc z)** reduz para a semântica de **suc z**, isto é, o número 1:

Demonstração. Temos pela definição que $\mathcal{S}[\![\text{isz } z]\!] = \top$, e consequentemente que $\mathcal{S}[\![\text{not isz } z]\!] = \bot$. Assim, $\mathcal{S}[\![\text{case } (\text{not } (\text{isz } z))(\text{suc } (\text{suc } z))(\text{suc } z)]\!]$ é igual a $\mathcal{S}[\![\text{suc z}]\!]$, que é de fato o número 1. □

2.1.1.3 Semântica Axiomática

Na Semântica Axiomática busca-se prover um sistema lógico que capture as propriedades essenciais dos programas em formas de asserções. Ao invés de definir o comportamento dos programas para então estudar suas propriedades, a semântica axiomática é formada por um conjunto de regras que permite deduzir propriedades sobre um certo programa (NIELSON; NIELSON, 2007). Neste trabalho, não será abordada esse tipo de semântica já que esta é utilizada para demonstrar propriedades sobre programas em linguagens imperativas, o que está fora do escopo deste trabalho.

2.2 Trabalhos Relacionados

O trabalho mais relacionado à este é o de Filinski (2001), que propõe um algoritmo de Normalização por Avaliação (NbE) para o λ-cálculo computacional. A normalização por avaliação apela para a semântica denotacional para produzir formas normais. O algoritmo do trabalho mencionado, no entanto, baseia-se em uma interpretação *quasi-static* e não foi formalizado em um assistente de provas. Também é importante ressaltar que a semântica denotacional nesse

caso está embutida nas definições do algoritmo, e seria bastante complicado extraí-la para outros propósitos que não utilizem a normalização por avaliação.

Kovács (2017) implementa um algoritmo de Normalização por Avaliação para o λ-cálculo simplesmente tipado, e prova sua corretude e completude através de um modelo de Kripke, em Agda. Enquanto Kovács formaliza a semântica do λ-cálculo simplesmente tipado para a implementação do NbE e prova a consistência e completude de suas formalizações, este trabalho apenas define a semântica denotacional para o λ-cálculo computacional.

Por fim, falando de outros trabalhos sobre o λ-cálculo computacional, o artigo de Benton, Bierman e Paiva (1998) deriva um sistema de lógica do referido sistema, através do Isomorfismo de Curry-Howard. Também apresenta de maneira mais elegante a sintaxe do sistema, que é usada neste trabalho.

3 Agda: Uma Visão Geral

Agda é uma linguagem de programação funcional com tipos dependentes. Seu sistema de tipos estende o da Teoria de Tipos de Martin-Löf com módulos e registros. Também é um assistente de provas pelo Isomorfismo de Curry-Howard (LOPES, 2018). Tendo sua sintaxe inspirada em Haskell, Agda foi criada através da tese de doutorado de Norell (2007), sendo atualmente desenvolvida com o apoio da comunidade *open-source*. Este capítulo apresenta alguns conceitos sobre Agda e exemplos. É assumido conhecimento básico sobre programação funcional.

3.1 Tipos de Dados e Tipos Dependentes

De acordo com Scott (2000), valores possuem o *status* de primeira classe numa linguagem de programação quando podem ser utilizados como parâmetros e como retorno, e também quando podem ser atribuídos à variáveis e armazenados em estruturas de dados. As linguagens de programação funcionais elevam as funções ao *status* de primeira classe. Em Agda, além das funções, os tipos também são valores de primeira classe. Todos os tipos básicos de Agda são do tipo Set. Para evitar o paradoxo de Russell, Agda possui uma hierarquia infinita de tipos, em que Set é do tipo Set_1, que por sua vez é do tipo Set_2, e assim por diante. Abaixo segue o clássico exemplo da definição de números naturais e a operação de soma em notação de Peano (1889) [1].

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
(suc m) + n = suc (m + n)
```

Observe que Agda aceita caracteres Unicode em nomes de tipos e funções. É possível criar operadores infixos com o uso de _.

Considere a definição abaixo de um tipo para listas de números naturais.

```
data ListNat : Set where
  []    : ListNat
  _::_ : ℕ → ListNat → ListNat
```

Em teorias de tipos não-dependentes, os tipos e os termos são entendidos como elementos separados e que interagem apenas para atribuir tipos a termos. Em teorias de tipos dependentes,

[1] Peano definiu 1 como o primeiro número natural, porém a maioria das formalizações modernas iniciam com 0.

os tipos podem depender de termos e expressar propriedades sobre eles (NORELL, 2007). Para reutilizar o código de todas as eventuais funções sobre a definição de lista acima, é possível definir listas polimórficas, usando os tipos dependentes.

```
data List (A : Set) : Set where
  []    : List A
  _::_ : A → List A → List A
```

Perceba como a definição de lista agora depende de um tipo ser passado como parâmetro. Para alcançar o máximo de generalidade possível, é possível definir listas para qualquer tipo de qualquer nível da hierarquia, usando o polimorfismo de nível. Parâmetros entre chaves são ditos ímplicitos.

```
data GenericList {ℓ : Level} (A : Set ℓ) : Set ℓ where
  []    : GenericList A
  _::_ : A → GenericList A → GenericList A
```

Em Haskell é bastante conhecida a função `head` que retorna a cabeça de uma lista, ou resulta num erro em tempo de execução quando executada sobre a lista vazia. Agda não admite esse tipo de comportamento e exige que toda função seja total. Assim, é preciso fazer uso do tipo Maybe para garantir que `head` seja total.

```
head : {A : Set} → List A → Maybe A
head []       = nothing
head (x :: xs) = just x
```

No entanto, é interessante usar os tipos dependentes para impor uma restrição à chamada de head, fazendo com que a própria assinatura da função impeça sua chamada sobre listas vazias. A definição de Vec abaixo possibilita isso.

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)

head' : ∀ {A n} → Vec A (suc n) → A
head' (x :: xs) = x
```

Nesta definição, o tipo Vec, além de polimórfico como as listas, depende também de um número natural para ser construído, que representa o tamanho do vetor. Assim, à medida que elementos são adicionados à um vetor, esse natural é automaticamente incrementado, pela definição do tipo. Como head' é um função definida para Vecs com tamanho maior que 0, é impossível executá-la sobre a lista vazia, e isso pode ser verificado em tempo de compilação.

3.2 Instance Arguments

De acordo com a documentação de Agda[2], *instance arguments* são um tipo especial de argumentos implícitos que são resolvidos por um algoritmo diferente do algoritmo padrão de unificação de tipos para argumentos implícitos. Tal algoritmo é chamado de *Instance Resolution* e suas etapas são explicadas na documentação.

Uma boa utilidade para os *instance arguments* de Agda é que eles podem ser usados para definir instâncias de classes de tipos. Em Agda, não há definição de classes de tipos tal como acontece em Haskell, mas pode-se utilizar registros para definí-las. Considere o seguinte exemplo de registro:

```
record Monoid (A : Set) : Set where
  field
    mempty : A
    _<>_ : A → A → A
```

Ele representa a classe de tipo dos Monóides, que são tipos que possuem um operador binário associativo e um elemento neutro com relação à esse operador. Uma instância de Monóide pode ser definida, utilizando *instance arguments*, para Listas usando o operador de concatenação:

```
instance
  ListMonoid : ∀ {A : Set} → Monoid (List A)
  ListMonoid = record { mempty = []; _<>_ = _++_ }
```

Com isso, mesmo que Agda não possua uma forma de criar classes de tipos, ainda é possível criar funções restritas a tipos com uma certa estrutura, e demonstrar que tipos possuem tal estrutura, quando é o caso.

3.3 Provas como Programas

De acordo com o isomorfismo de Curry-Howard, Agda pode ser utilizada como um assistente de provas. Como exemplo de uma prova, considere a prova de que $n + 0 \equiv n$:

```
+-idʳ : ∀ (n : ℕ) → n + zero ≡ n
+-idʳ zero = refl
+-idʳ (suc n) rewrite +-idʳ n = refl
```

A assinatura da função descreve, através do tipo dependente, a propriedade de que para todo natural, a soma dele com 0 é o próprio natural. Escrever uma implementação recursiva

[2] Disponível no endereço <https://agda.readthedocs.io/en/v2.6.1/language/instance-arguments.html>

para uma função com este tipo equivale a escrever uma prova por indução para a proposição equivalente. Os casamentos de padrão equivalem aos casos da prova, e a chamada recursiva após o `rewrite` equivale à hipótese de indução. Assim, essa propriedade é provada mostrando que ela é satisfeita para o zero e que quando é satisfeita por um natural qualquer, também será satisfeita pelo próximo natural, completando a prova por indução.

Ao contrário de Coq, Agda não possui táticas separadas para uso em provas, tendo as provas escritas em estilo de programação funcional. A maioria dos exemplos deste capítulo são reproduções dos exemplos dados por Lopes (2018) em sua dissertação de mestrado. Mais informações sobre Agda podem ser encontradas no livro de Wadler e Kokke (2019).

4 O Lambda Cálculo

O λ-cálculo é um sistema formal proposto por Alonzo Church em dois trabalhos, sendo um deles o que visava provar que o *Entscheidungsproblem* de Hilbert era indecidível (CHURCH, 1936). É também o sistema que Peter Landin utilizou para mostrar que uma linguagem complexa pode ser entendida formalizando suas partes essenciais em um sistema menor, e definindo as outras construções, que são apenas notações convenientes, em termos desse sistema (PIERCE, 2002). O λ-cálculo original de Church foi provado ser inconsistente por seus alunos Kleene e Rosser (1935), ou seja, qualquer fórmula poderia ser provada, independente do significado da interpretação de seus símbolos. Uma nova versão consistente, com tipagem simples e estática, foi desenvolvida pelo próprio Church (1940), e ficou conhecida como λ-cálculo simplesmente tipado. Existem vários sistemas baseados em diferentes versões do λ-cálculo, e a maioria desses sistemas serve de base teórica para algum recurso de linguagens de programação. Este capítulo apresenta as duas versões principais (tipado e não tipado) e o λ-cálculo computacional.

4.1 Lambda Cálculo Não Tipado

Um ponto chave na reusabilidade de código é a possibilidade de se definir funções e dar a elas um nome que será usado para executar seu código sobre um ou mais argumentos, ao invés de replicá-la exautivamente sempre que for necessário. Para Pierce (2002), o λ-cálculo captura essa noção de definir e aplicar funções em sua forma mais pura. Tudo é função nesse sistema, incluindo constantes e os argumentos e resultados de outras funções. A sintaxe do λ-cálculo não tipado é definida pela seguinte gramática livre de contexto:

$$t \rightarrow x \mid \lambda x.t \mid t\ t$$

Todo termo t do λ-cálculo é uma variável, representada por x, uma abstração, representada por $\lambda x.t$, ou uma aplicação, representada por $t\ t$.

Abstrações são formas anônimas de definir uma função, i.e. sem vinculá-la a um nome. Os argumentos, no entanto, são nomeados através das variáveis ligadas, como é o caso do x na função identidade $\lambda x.x$. Renomear uma variável ligada não altera o significado/comportamento de uma função. Por exemplo, $\lambda y.y$ ainda é a função identidade. Esse conceito é conhecido como α-equivalência, que consiste em reconhecer como iguais termos a menos de renomear variáveis ligadas. Mas é preciso tomar cuidado para que o novo nome não coincida com uma variável livre. Renomear x para z na função $\lambda x.z$ a transformaria na função identidade, quando ela originalmente era uma função que ignorava o argumento e retornava o valor da variável livre z. É possivel escrever termos do λ-cálculo sem necessariamente dar nome às variáveis ligadas. Isso

é explicado melhor na Subseção 4.2.1.1.

Aplicações no λ-cálculo se tratam de substituir todas as ocorrências de uma variável ligada no corpo da abstração por um termo dado como parâmetro, e então remover o "cabeçalho" da abstração correspondente. Por exemplo, a aplicação $(\lambda x.xx)\ (\lambda z.z)$ resulta na aplicação $(\lambda z.z)(\lambda z.z)$, que por sua vez resulta em $\lambda z.z$. Esse processo é chamado de β-redução. Novamente, é preciso tomar cuidado para que variáveis livres que serão aplicadas à uma abstração não se confundam com outras variáveis ligadas presentes no corpo da abstração. Por exemplo, a aplicação $(\lambda x.xyz)\,(\lambda y.yz)$ faria com que o y livre se tornasse ligado, se a redução fosse feita ingenuamente. Nesse caso, precisa-se valer da α-equivalência para renomear o y ligado e assim eliminar os possíveis conflitos. Note que não há garantia que uma β-redução enventualmente produzirá um termo que não pode mais ser aplicado. Termos como $(\lambda x.xx)(\lambda x.xx)$ e $(\lambda x.xxx)(\lambda x.xxx)$ não podem ser reduzidos em um número finito de passos.

Apesar de ser simples, o sistema é bastante poderoso, sendo capaz de representar toda função computável. A versão não tipada apresentada também é capaz de representar paradoxos, e portanto, é inconsistente (KLEENE; ROSSER, 1935).

O λ-cálculo pode ser facilmente definido em Agda com a abordagem intrínseca, como mostrado abaixo. Esse conceito é explicado melhor na Seção 4.2.1. Note que como este λ-cálculo não é tipado, isso é o mesmo que dizer que todos os termos possuem o mesmo tipo (WADLER; KOKKE, 2019).

```
data Type : Set where
  * : Type

data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context

data _∋_ : Context → Type → Set where
  here  : ∀ {Γ A} → Γ , A ∋ A
  there : ∀ {Γ A B} → Γ ∋ A → Γ , B ∋ A

data _⊢_ : Context → Type → Set where
  var : ∀ {Γ A} → Γ ∋ A → Γ ⊢ A
  abs : ∀ {Γ} → Γ , * ⊢ * → Γ ⊢ *
  app : ∀ {Γ} → Γ ⊢ * → Γ ⊢ * → Γ ⊢ *
```

A formalização acima é uma adaptação da apresentada por Wadler e Kokke (2019) em seu livro. Na formalização do λ-cálculo simplesmente tipado, na próxima Seção, será possível ver com mais clareza como o sistema de tipos se relaciona com a estrutura dos termos na abordagem intrínseca. Mais informações sobre o λ-cálculo não tipado, incluindo a semântica e estratégias de avaliação, podem ser encontradas no livro de Pierce (2002).

4.2 Lambda Cálculo Simplesmente Tipado

Considere a linguagem de exemplo definida na Seção 2.1. Mesmo que a semântica tenha sido definida com cuidado, há programas sintaticamente válidos que não teriam uma interpretação semântica. A gramática permite que se construa `not z` ou `isz f`, mas não há significado definidos para essas construções. O motivo é que `not` foi pensado para ser o equivalente ao operador de negação $\neg$ da Lógica, e por isso não faz sentido falar de `not` quando aplicado em números, a menos que se defina uma correspondência entre números e booleanos, como faz a linguagem C/C++. Dessa forma, um programa sintaticamente válido poderia ficar "preso" ao ser executado usando as definições de semântica operacional. Acrescentar a noção de tipos naquela linguagem permite restringir os programas sintaticamente válidos, e com isso, permitir que todo programa válido possua significado e jamais fique "preso", uma propriedade conhecida como **Progresso**. Assim como na linguagem de exemplo, a inclusão de tipos no λ-cálculo sacrifica um pouco da sua flexibilidade mas proíbe a aplicação de funções sobre argumentos incorretos (SELINGER, 2008).

4.2.1 Abordagens para o lambda cálculo tipado

Existem duas abordagens fundamentais para o λ-cálculo: a abordagem intrínseca e a extrínseca. Na primeira, termos são definidos de acordo com seus tipos, de maneira simultânea. Na segunda, termos e seus tipos são definidos separadamente. A abordagem intrínseca é de grande utilidade nos assistentes de provas, possibilitando formalizações mais compactas. Tal abordagem é baseada nos índices de De Bruijn, apresentado na Seção 4.2.1.1, logo abaixo. As abordagens intrínseca e extrínseca também são chamadas, respectivamente, de *Church style* e *Curry style* (WADLER; KOKKE, 2019).

4.2.1.1 Índices de De Bruijn

A renomeação de variáveis ligadas nos processos mencionados anteriormente podem se tornar bastante problemáticas em expressões longas e de estrutura complexa. Uma notação alternativa foi desenvolvida por De Bruijn (1972), em que variáveis ligadas são representadas por números indicando a distância delas até o seu λ correspondente. Dessa forma, o uso de nomes é desnecessário para variáveis ligadas. Por exemplo, a identidade $\lambda x.x$ pode ser reescrita como $\lambda.1$, ocultando-se agora o nome da variável na cabeça da função, e substituindo-se a sua ocorrência no corpo pela indicação de que seu λ correspondente é o primeiro ao caminhar-se para trás na hierarquia de escopo. O termo $\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.zx)$ pode ser reescrito como $\lambda(\lambda.1(\lambda.1))(\lambda.2\ 1)$, facilitando todas as eventuais aplicações envolvendo esse termo, já que não será mais preciso se preocupar com os nomes das variáveis.

4.2.2 Sintaxe

O λ-cálculo simplesmente tipado (STLC) estende o sistema original para incluir tipos simples. Usa-se a notação $e : \tau$ para representar que uma expressão e tem tipo τ. A sintaxe desse sistema pode ser definida pela gramática a seguir.

$$
\begin{array}{lcl}
t & \rightarrow & x \;\mid\; \lambda x : \tau.t \;\mid\; t\,t \\
\tau & \rightarrow & \tau \rightarrow \tau \\
\Gamma & \rightarrow & \varnothing \;\mid\; \Gamma, x : \tau
\end{array}
$$

Uma lista de variavéis e/ou suposições sobre elas é chamada de Contexto. Usa-se a notação $\Gamma, e_1 \vdash e_2$ para representar que a expressão e_2 pode ser obtida a partir do contexto Γ, estendido com uma suposição sobre e_1. A semântica e o sistema de tipos podem ser encontrados no livro de Pierce (2002). O sistema de tipos é reproduzido abaixo, para efeitos de comparação com a definição em Agda.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{(var)} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1.t : \tau_1 \rightarrow \tau_2}\ \text{(abs)}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ \text{(app)}$$

A regra (var) define que o tipo de uma variável livre é determinado pelo contexto Γ. A regra (abs) define que uma abstração tem tipo $\tau_1 \rightarrow \tau_2$ quando é possível obter o termo de tipo τ_2 ao estender o contexto Γ com a variável de tipo τ_1. Por fim, a regra (app) define que um termo de tipo $\tau_1 \rightarrow \tau_2$ é reduzido a um termo de tipo de τ_2 quando aplicado em um termo de tipo τ_1. Na abordagem intrínseca, o sistema de tipos praticamente se torna a definição do sistema.

Um exemplo de derivação de tipos para $\{y : bool, z : int\} \vdash (\lambda x.y)\ z$ é apresentado logo abaixo, demonstrando que tal expressão tem tipo $bool$:

$$\dfrac{\dfrac{\dfrac{y : bool \in \{y : bool, z : int, x : int\}}{\{y : bool, z : int, x : int\} \vdash y : bool}\ \text{(var)}}{\{y : bool, z : int\} \vdash \lambda x.y : int \rightarrow bool}\ \text{(abs)} \qquad \dfrac{z : int \in \{y : bool, z : int\}}{\{y : bool, z : int\} \vdash z : int}\ \text{(var)}}{\{y : bool, z : int\} \vdash (\lambda x.y)\ z : bool}\ \text{(app)}$$

Note como as regras de tipagem do STLC se reforçam na formalização em Agda apresentada abaixo. As definições de contexto e pertinência são ocultadas por serem as mesmas do λ-cálculo não tipado.

```
data Type : Set where
  γ       : Type
  _⇒_ : Type → Type → Type

data _⊢_ : Context → Type → Set where
  var : ∀ {Γ A} → Γ ∋ A → Γ ⊢ A
  abs : ∀ {Γ τ₁ τ₂} → Γ , τ₁ ⊢ τ₂ → Γ ⊢ τ₁ ⇒ τ₂
  app : ∀ {Γ τ₁ τ₂} → Γ ⊢ τ₁ ⇒ τ₂ → Γ ⊢ τ₁ → Γ ⊢ τ₂
```

A definição apresenta um tipo que não está definido na gramática apresentada, o tipo γ. Esse tipo representa, na verdade, qualquer tipo básico (*ground type*) dos termos. O STLC não define nenhum tipo básico específico, sendo eles dependentes do contexto da aplicação do sistema. Além de Progresso, Pierce (2002) define uma outra propriedade interessante a ser investigada em sistemas tipados. Essa propriedade, chamada de **Preservação**, diz respeito a se todo termo bem-tipado produz apenas outros termos bem tipados ao serem reduzidos. Tanto Progresso quanto Preservação podem ser provados por indução. Mais detalhes sobre o STLC podem ser encontrados também nas notas de aula de Selinger (2008).

4.3 Lambda Cálculo Computacional

Ao se utilizar as versões mais elementares do λ-cálculo para mostrar equivalência entre programas, é necessário considerar programas como funções totais. Tal consideração pode representar um obstáculo na aplicação dos resultados teóricos, o que inspirou Moggi (1988) a criar um sistema que poderia ser usado para raciocinar sobre equivalência entre programas. Esse sistema, desenvolvido usando Teoria das Categorias, é conhecido como λ-cálculo computacional (CLC). Mais especificamente, Moggi usa uma semântica categórica (e portanto, denotacional) de **computação**, o que se reflete no operador `T(.)` que possui a estrutura de uma mônada. Uma extensão do sistema pode ser usada para derivar um sistema de Lógica Intuicionista Proposicional com um operador similar ao de *possibilidade* na Lógica Modal (BENTON; BIERMAN; PAIVA, 1998).

4.3.1 Semântica Categórica de Computação

Moggi (1991) define **modelo de computação**, do ponto de vista categórico, como uma mônada M sobre uma categoria C. A idéia por trás dessa definição é que programas que recebem como parâmetro valores do tipo A, e retornam valores do tipo B, são como "funções" de A para $T\ B$ (o conjunto das computações de tipo B). Assim, essa semântica independe de qualquer modelo computacional específico. Computações não-deterministas, com efeitos colaterais, etc. se encaixam facilmente nessa definição geral.

4.3.2 Sintaxe e Formalização em Agda

Uma possível definição da sintaxe do CLC é a seguinte:

$$
\begin{array}{rcl}
t & \to & x \mid \lambda x : \tau . t \mid t\,t \mid * \mid (t,t) \\
 & & \mid \mathtt{inl}\ t \mid \mathtt{inr}\ t \mid \mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}\ t \to t \mid \mathtt{inr}\ t \to t \\
 & & \mid \mathtt{fst}\ t \mid \mathtt{snd}\ t \mid \mathtt{val}\ t \mid \nabla_\tau\, t \mid \mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ t \\
\tau & \to & \tau \to \tau \mid 1 \mid 0 \\
 & & \mid \tau \times \tau \mid \tau + \tau \mid T\,\tau \\
\Gamma & \to & \varnothing \mid \Gamma, x : \tau
\end{array}
$$

Pode-se entender o CLC como uma extensão do STLC. Os operadores $\times$ e $+$ correspondem, respectivamente, à produto e soma de tipos. Dessa forma, `fst` e `snd` correspondem às projeções e `inl` e `inr` correspondem às injeções. O construtor `T` corresponde à mônada. O operador ∇ modela construções monádicas como computações não definidas ou que não terminam. Tem-se o `undefined` de Haskell como exemplo de ∇.

O sistema de tipos, apresentado abaixo, está de acordo com o trabalho de Benton, Bierman e Paiva (1998). Ele é associado a uma dedução natural que os autores chamam de lógica CL, que não será apresentada.

$$
\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A . e : A \to B}
$$

$$
\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B} \qquad \frac{}{\Gamma \vdash * : 1}
$$

$$
\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathrm{val}(e) : TA}
$$

$$
\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathrm{fst}(e) : A} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathrm{snd}(e) : B}
$$

$$
\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathrm{inl}_{A+B}(e) : A + B} \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathrm{inr}_{A+B}(e) : A + B}
$$

$$
\frac{\Gamma \vdash e_1 : T\,A \qquad \Gamma, x : A \vdash e_2 : T\,B}{\Gamma \vdash \mathrm{let}\ x \Leftarrow e_1\ \mathrm{in}\ e_2 : T\,B} \qquad \frac{\Gamma \vdash e : 0}{\Gamma \vdash \nabla_A\, e : A}
$$

$$
\frac{\Gamma \vdash e : A + B \qquad \begin{array}{c}\Gamma, e_1 : A \vdash e_1' : C \\ \Gamma, e_2 : B \vdash e_2' : C\end{array}}{\Gamma \vdash \mathrm{case}\ e\ \mathrm{of}\ \mathrm{inl}(e_1) \to e_1' \mid \mathrm{inr}(e_2) \to e_2' : C}
$$

É importante ressaltar que algumas definições para o STLC incluem produto e soma

de tipos da mesma forma que o CLC, como por exemplo as notas de aula de Selinger (2008). Apresenta-se a formalização em Agda para o CLC, trecho a trecho e relacionando-os com as regras do sistema de tipos. Começa-se pela definição dos tipos.

```
data Type : Set where
  γ     : Type
  Z     : Type -- Zero
  I     : Type -- One
  _×_   : Type → Type → Type
  _+_   : Type → Type → Type
  T     : Type → Type
  _⇒_   : Type → Type → Type
```

O tipo γ corresponde a um ou mais tipos básicos quaisquer, Z e I correspondem, respectivamente, aos tipos 0 e 1 do sistema de tipos. O tipo T corresponde à mônada. O restante são os operadores de produto ($\times$), soma ($+$) e o de construtor de tipos funcionais ($\Rightarrow$).

A definição dos termos seguem o sistema de tipos previamente apresentado. Assim como no STLC, temos os três construtores de termos mais "básicos"—variáveis, abstrações e aplicações.

```
data _⊢_ : Context → Type → Set where
```

```
var : ∀ {Γ A} → Γ ∋ A → Γ ⊢ A
```

$$\frac{}{\Gamma, x : A \vdash x : A}$$

```
abs : ∀ {Γ A B} → Γ , A ⊢ B → Γ ⊢ A ⇒ B
```

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \to B}$$

```
app : ∀ {Γ A B} → Γ ⊢ A ⇒ B
                → Γ ⊢ A
                → Γ ⊢ B
```

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

Tal como verdadeiro pode ser sempre introduzido num sistema de dedução natural, sempre é possível derivar o tipo 1 a partir de um contexto qualquer (inclusive vazio) no CLC. Já o tipo 0 só pode ser derivado de programas que não terminam ou de outros casos particulares, e com isso tem-se o termo ∇ capaz de construir qualquer tipo.

```
top : ∀ {Γ} → Γ ⊢ I
```

$$\frac{}{\Gamma \vdash * : 1}$$

```
bot : ∀ {Γ A} → Γ ⊢ Z → Γ ⊢ A
```

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \nabla_A\, e : A}$$

O produto de termos pode ser introduzido a partir de dois termos deriváveis do contexto, e também é possível fazer projeções do primeiro ou o segundo termo de um produto.

$$\mathsf{prod} : \forall\,\{\Gamma\ A\ B\} \to \Gamma \vdash A \to \Gamma \vdash B \to \Gamma \vdash A \times B \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}$$

$$\mathsf{fst} : \forall\,\{\Gamma\ A\ B\} \to \Gamma \vdash A \times B \to \Gamma \vdash A \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathrm{fst}(e) : A}$$

$$\mathsf{snd} : \forall\,\{\Gamma\ A\ B\} \to \Gamma \vdash A \times B \to \Gamma \vdash B \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathrm{snd}(e) : B}$$

De maneira análoga, a soma de tipos pode ser introduzida a partir de injeções à esquerda ou direita, e também pode ser reduzida a um termo que é derivável a partir de cada termo desta soma.

$$\mathsf{inl} : \forall\,\{\Gamma\ A\ B\} \to \Gamma \vdash A \to \Gamma \vdash A + B \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathrm{inl}_{A+B}(e) : A + B}$$

$$\mathsf{inr} : \forall\,\{\Gamma\ A\ B\} \to \Gamma \vdash B \to \Gamma \vdash A + B \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathrm{inr}_{A+B}(e) : A + B}$$

$$\begin{aligned}\mathsf{case} : \forall\,\{\Gamma\ A\ B\ C\} &\to \Gamma \vdash A + B \\ &\to \Gamma\ ,\ A \vdash C \\ &\to \Gamma\ ,\ B \vdash C \\ &\to \Gamma \vdash C\end{aligned} \qquad \frac{\Gamma \vdash e : A + B \qquad \begin{array}{c}\Gamma, e_1 : A \vdash e_1' : C \\ \Gamma, e_2 : B \vdash e_2' : C\end{array}}{\Gamma \vdash \mathrm{case}\ e\ \mathrm{of}\ \mathrm{inl}(e_1) \to e_1' \mid \mathrm{inr}(e_2) \to e_2' : C}$$

Por fim, tem-se os operadores da mônada: o val que apenas encapsula um valor dentro do operador monádico e o bind para avaliar funções com o uso desses valores encapsulados. Esses operadores correspondem ao `return` e >>= de Haskell.

$$\mathsf{val} : \forall\,\{\Gamma\ A\} \to \Gamma \vdash A \to \Gamma \vdash \mathsf{T}\ A \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathrm{val}(e) : TA}$$

$$\begin{aligned}\mathsf{bind} : \forall\,\{\Gamma\ A\ B\} &\to \Gamma \vdash \mathsf{T}\ A \\ &\to \Gamma\ ,\ A \vdash \mathsf{T}\ B \\ &\to \Gamma \vdash \mathsf{T}\ B\end{aligned} \qquad \frac{\Gamma \vdash e_1 : T\ A \qquad \Gamma, x : A \vdash e_2 : T\ B}{\Gamma \vdash \mathrm{let}\ x \Leftarrow e_1\ \mathrm{in}\ e_2 : T\ B}$$

Existem várias outras versões de λ-cálculo que podem ser encontradas na literatura. Exemplos incluem o λ-cálculo polimórfico e o λ-cálculo gradualmente tipado. Mais informações sobre o λ-cálculo computacional são encontradas nos artigos de Moggi (1988) e Benton, Bierman e Paiva (1998), já mencionados acima.

5 Modelo Padrão

O modelo padrão de semântica denotacional é o mapeamento direto das construções sintáticas para objetos matemáticos. Kovács (2017), por exemplo, utiliza um modelo de Kripke no seu trabalho. Este capítulo apresenta o modelo padrão para o STLC e o adapta para servir para o CLC. Este Capítulo, assim como os Capítulos 3 e 4, são arquivos Literate Agda verificados utilizando Agda versão 2.6.1 com a biblioteca padrão versão 1.4. O código pode ser encontrado no seguinte repositório do GitHub: <https://github.com/mayconamaro/clc-denotational-agda>.

5.1 Lambda Cálculo Simplesmente Tipado

Por se tratar de uma abordagem intrínseca, primeiro deve-se formalizar os tipos.

```
⟦_⟧ : Type → Set
⟦ γ ⟧       = ⊥
⟦ A ⇒ B ⟧ = ⟦ A ⟧ → ⟦ B ⟧
```

Não há uma forma sistemática de mapear tipos, e considerando que o *tipo básico* é na verdade um coringa para representar tipos primários que surgem na aplicação do sistema, um mapeamento possível é definir o tipo γ como o tipo vazio $\bot$—um conjunto vazio. O tipo de funções do CLC é associado ao construtor de tipos funcionais de Agda.

No STLC, uma possível semântica consiste em definir contextos como listas de tipos. Uma lista pode ser definida como tuplas aninhadas, uma perspectiva um pouco mais compatível com a Álgebra.

```
⟦_⟧C : Context → Set
⟦ ∅ ⟧C     = ⊤
⟦ Γ , A ⟧C = ⟦ A ⟧ Prod.× ⟦ Γ ⟧C
```

A relação de pertinência entre uma variável se mantém como a busca pela variável no contexto. A relação definida constrói a evidência da pertinência, ao invés de apenas decidir.

```
data _∋_ : Context → Type → Set where
  here  : ∀ {Γ A} → Γ , A ∋ A
  there : ∀ {Γ A B} → Γ ∋ A → Γ , B ∋ A
```

Assim, a semântica de uma variável é a função que dada o contexto que contém tal variável, retorna a semântica de seu tipo. `Prod` é um *alias* para o módulo `Data.Product` de Agda, e a distinção é feita para evitar confusões entre a notação do módulo e a sintaxe do STLC.

```
semVar : ∀ {A Γ} → Γ ∋ A → ⟦ Γ ⟧C → ⟦ A ⟧
semVar (here)    (v Prod., _) = v
semVar (there v) env          = semVar v (Prod.proj₂ env)
```

Os termos são interpretados sob a mesma noção matemática de variável, definição de função e aplicação de função.

```
eval : ∀ {A Γ} → Γ ⊢ A → ⟦ Γ ⟧C → ⟦ A ⟧
eval (var x)     env = semVar x env
eval (abs x)     env = λ z → eval x (z Prod., env)
eval (app x x₁) env = (eval x env) (eval x₁ env)
```

A função eval avalia variáveis utilizando a semântica definida previamente, avalia abstrações criando funções anônimas correspondentes em Agda. Tal função, ao receber um parâmetro z, o adiciona ao contexto e executa eval com o termo usando esse contexto estendido. eval avalia aplicações executando uma aplicação de função de Agda entre os dois termos já interpretados.

5.2 Lambda Cálculo Computacional

Para a adaptação do modelo padrão na Seção anterior, é preciso repensar os tipos:

```
⟦_⟧ : Type → Set
⟦ I ⟧   = ⊤
⟦ Z ⟧ = ⊥
⟦ γ ⟧ = ⊥ -- tipo básico coringa
⟦ A × B ⟧ = ⟦ A ⟧ Prod.× ⟦ B ⟧
⟦ A + B ⟧ = ⟦ A ⟧ ⊎ ⟦ B ⟧
⟦ A ⇒ B ⟧ = ⟦ A ⟧ → ⟦ B ⟧
⟦ T A ⟧     = M ⟦ A ⟧
```

O λ-cálculo computacional possui outros tipos além do coringa para tipos básicos. O tipo do produto e do coproduto podem ser mapeados para o produto e soma de tipos de Agda (que também podem ser vistos como produto cartesiano e união da Teoria de Conjuntos). O construtor T pode ser mapeado para qualquer tipo que suporte as operações monádicas `return` e `>>=`, e assim é definido com o auxílio de um *instance argument*.

A semântica de contextos e pertinência seguem iguais ao do STLC, e por isso são omitidas. Para a semântica de termos, cobriremos caso a caso.

```
eval : ∀ {t Γ} → Γ ⊢ t → ⟦ Γ ⟧C → ⟦ t ⟧
```

O primeiro caso é semântica de variável, que é idêntica ao STLC e permanece sem modificações.

```
eval (var v) env = semVar v env
```

O termo `top` representa a derivação do tipo 1 (modelado como I). Assim como a semântica de 1 é definida como sendo o valor $\top$ da lógica, a semântica do termo `top` diz respeito à introdução do verdadeiro na lógica, uma propriedade que se pode invocar utilizando o tt da biblioteca padrão de Agda.

```
eval (top) env = tt
```

O termo `bot` representa o operador ∇ construindo um tipo qualquer a partir de um termo de tipo 0 (modelado como Z). Associando o tipo 0 ao $\bot$ da lógica, este termo equivale à regra de eliminação do falso da lógica, que está implementada na biblioteca padrão de Agda sob o nome ⊥-elim.

```
eval (bot e) env = ⊥-elim (eval e env)
```

A semântica de produto de tipos é associada ao produto de tipos da biblioteca padrão de Agda, que também conta com as projeções—utilizadas pelos termos `fst` e `snd`. `Prod.`, significa que a vírgula é um operador para construção de tuplas definido no módulo Data.Product.

```
eval (prod e e') env = eval e env Prod., eval e' env
eval (fst e) env = Prod.proj₁ (eval e env)
eval (snd e) env = Prod.proj₂ (eval e env)
```

A abstração e aplicação são idênticas às definições já apresentadas para o STLC, sem modificações.

```
eval (abs e) env = λ z → eval e (z Prod., env)
eval (app e e') env = (eval e env) (eval e' env)
```

A soma de tipos é associada à definição de soma de tipos da biblioteca padrão de Agda. Tal definição conta com as injeções e um operador `[_,_]` para a eliminação da disjunção. Esse operador é renomeado como ⊎-elim.

```
eval (inl e) env = inj₁ (eval e env)
eval (inr e) env = inj₂ (eval e env)
eval (case e e' e'') env = ⊎-elim (λ v → eval e' (v Prod., env))
                                  (λ v → eval e'' (v Prod., env))
                                  (eval e env)
```

Por fim, a semântica dos termos `val` e `bind` são associados às construções que definem as mônadas: `return` e >>=. Assim como as mônadas, `val` e `return` são capazes de lidar com

computações contendo efeitos colaterais—exemplificando um dos grandes diferenciais do CLC com relação aos λ-cálculos mais elementares.

```
eval (val e) env = return (eval e env)
eval (bind e e') env = eval e env >>= λ v → eval e' (v Prod., env)
```

Isto conclui a formalização da semântica denotacional do λ-cálculo computacional.

6 Considerações Finais

6.1 Conclusão

Projetistas de linguagens de programação constantemente exploram novos recursos para serem incorporados às linguagens, para oferecer mais flexibilidade aos programadores sem sacrificar desempenho, segurança e produtividade. Esses recursos precisam de uma base teórica sólida antes de serem implementados, e frequentemente são representados através de um sistema menor—o λ-cálculo—capaz de capturar a essência da linguagem e separá-las de construções secundárias. A definição rigorosa dos significados dos símbolos e sentenças dessas linguagens é fundamental para o seu estudo. A semântica denotacional é útil por associar sintaxe de código a objetos matemáticos, permitindo assim o uso de resultados da matemática para assegurar propridades de interesse sobre programas. Neste trabalho, apresentamos o λ-cálculo computacional e uma formalização, verificada pelo assistente Agda, do modelo padrão de sua semântica denotacional.

6.2 Trabalhos Futuros

Uma possibilidade de continuação deste trabalho é a de aplicar essa semântica para a definição de um algoritmo de Normalização por Avaliação como proposto por Kovács (2017), adaptando o modelo para as provas de corretude e completude, já que o modelo de Kripke usado por Kovács pode não ser suficiente para o CLC, sendo necessária a investigação.

Referências

BENTON, P. N.; BIERMAN, G. M.; PAIVA, V. C. V. de. Computational types from a logical perspective. *Journal of Functional Programming*, Cambridge University Press, v. 8, n. 2, p. 177–193, 1998.

CHURCH, A. An unsolvable problem of elementary number theory. *American journal of mathematics*, JSTOR, v. 58, n. 2, p. 345–363, 1936.

CHURCH, A. A formulation of the simple theory of types. *The journal of symbolic logic*, Cambridge University Press, v. 5, n. 2, p. 56–68, 1940.

De Bruijn, N. G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In: NORTH-HOLLAND. *Indagationes Mathematicae (Proceedings)*. [S.l.], 1972. v. 75, n. 5, p. 381–392.

FILINSKI, A. Normalization by evaluation for the computational lambda-calculus. In: SPRINGER. *International Conference on Typed Lambda Calculi and Applications*. [S.l.], 2001. p. 151–165.

KAHN, G. Natural semantics. In: SPRINGER. *Annual symposium on theoretical aspects of computer science*. [S.l.], 1987. p. 22–39.

KLEENE, S. C.; ROSSER, J. B. The inconsistency of certain formal logics. *Annals of Mathematics*, JSTOR, p. 630–636, 1935.

KOVÁCS, A. *A Machine-Checked Correctness Proof of Normalization by Evaluation for Simply Typed Lambda Calculus*. Dissertação (Mestrado) — Eötvös Loránd University, 2017.

LOPES, R. *Certified Derivative-based Parsing of Regular Expressions*. Dissertação (Mestrado) — Universidade Federal de Ouro Preto, 2018.

MILEWSKI, B. *Category theory for programmers*. [S.l.]: Blurb, 2018.

MOGGI, E. *Computational lambda-calculus and monads*. [S.l.]: University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1988.

MOGGI, E. Notions of computation and monads. *Information and computation*, Elsevier, v. 93, n. 1, p. 55–92, 1991.

NIELSON, H. R.; NIELSON, F. *Semantics with applications: an appetizer*. [S.l.]: Springer Science & Business Media, 2007.

NORELL, U. *Towards a practical programming language based on dependent type theory*. Tese (Doutorado) — Chalmers University of Technology and Göteborg University, 2007.

PEANO, G. *Arithmetices principia: nova methodo exposita*. [S.l.]: Fratres Bocca, 1889.

PIERCE, B. C. *Types and programming languages*. [S.l.]: MIT press, 2002.

PLOTKIN, G. D. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, v. 60-61, p. 17–139, 2004.

SCOTT, M. L. *Programming Language Pragmatics*. [S.l.]: Morgan Kaufmann, 2000.

SELINGER, P. Lecture notes on the lambda calculus. *arXiv preprint arXiv:0804.3434*, 2008.

WADLER, P.; KOKKE, W. *Programming Language Foundations in Agda*. [S.l.: s.n.], 2019. Available at <http://plfa.inf.ed.ac.uk/>.