UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

HUGO CARVALHO DE PAULA

MATEMÁTICA FEITA NO COMPUTADOR: ESTUDOS SOBRE UMA COLEÇÃO DE SISTEMAS FORMAIS, COM UMA INVESTIGAÇÃO SOBRE ESTRUTURAS INDUTIVAS, E A SEPARAÇÃO ENTRE LÓGICA E AUTOMAÇÃO EM ASSISTENTES DE PROVA

FORTALEZA

2018

HUGO CARVALHO DE PAULA

MATEMÁTICA FEITA NO COMPUTADOR: ESTUDOS SOBRE UMA COLEÇÃO DE SISTEMAS FORMAIS, COM UMA INVESTIGAÇÃO SOBRE ESTRUTURAS INDUTIVAS, E A SEPARAÇÃO ENTRE LÓGICA E AUTOMAÇÃO EM ASSISTENTES DE PROVA

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação.

Orientador: Prof. Dr. Carlos Eduardo Fisch de Brito

FORTALEZA

2018

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P347m Paula, Hugo Carvalho de.
Matemática feita no computador: Estudos sobre uma coleção de sistemas formais, com uma investigação sobre estruturas indutivas, e a separação entre lógica e automação em assistentes de prova / Hugo Carvalho de Paula. – 2018.
67 f.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2018.
Orientação: Prof. Dr. Carlos Eduardo Fisch de Brito.

1. Assistentes de prova. 2. Computação. 3. Fundamentos da Matemática. 4. Teoria dos Tipos. 5. Lógica. I. Título.

CDD 005

HUGO CARVALHO DE PAULA

MATEMÁTICA FEITA NO COMPUTADOR: ESTUDOS SOBRE UMA COLEÇÃO DE SISTEMAS FORMAIS, COM UMA INVESTIGAÇÃO SOBRE ESTRUTURAS INDUTIVAS, E A SEPARAÇÃO ENTRE LÓGICA E AUTOMAÇÃO EM ASSISTENTES DE PROVA

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Carlos Eduardo Fisch de Brito (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Ruy José Guerra Barretto de Queiroz
Universidade Federal de Pernambuco (UFPE)

Prof. Dr. João Fernando Lima Alcântara
Universidade Federal do Ceará (UFC)

Prof. Dr. Pablo Mayckon Silva Farias
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

Ao Prof. Dr. Carlos Eduardo Fisch de Brito por me orientar em minha dissertação de mestrado.

Ao Doutorando em Engenharia Elétrica, Ednardo Moreira Rodrigues, e seu assistente, Alan Batista de Oliveira, aluno de graduação em Engenharia Elétrica, pela adequação do *template* utilizado neste trabalho para que o mesmo ficasse de acordo com as normas da biblioteca da Universidade Federal do Ceará (UFC).

À minha família como um todo, por tudo que já fizeram, e pela clareza de exemplo que são.

Aos professores e colegas, que me acompanharam nessa jornada.

À Fundação Cearense de Apoio ao Desenvolvimento (Funcap), pelo financiamento da minha pesquisa de mestrado, através de bolsa de estudos.

RESUMO

Este trabalho desenvolve um estudo de uma sequência de sistemas formais, com o propósito de investigar a relação entre computação e matemática, e apresenta uma possível nova direção para o desenvolvimento de assistentes de prova. O ponto de partida é o Cálculo Lambda Simplesmente Tipado, e sucessivas extensões são tratadas, culminando no CoqMT. O trabalho trata apenas de alguns aspectos de cada sistema, como a matemática neles realizável, e como que computação neles se manifesta. Embora existam discussões independentes de cada um desses sistemas, aqui há uma ênfase na visão desses sistemas como uma sequência propriamente dita, o que possibilita uma discussão clara de ideias recorrentes. Com base em tais ideias, a dissertação apresenta o que chamamos de "estruturas indutivas", uma generalização de conceitos encontrados no Cálculo das Construções Indutivas. A dissertação conclui com exemplos que ilustram o uso de computação na presença dessas estruturas, e esboça uma implementação com base em trabalhos pré-existentes da literatura.

Palavras-chave: Assistentes de Prova. Computação. Fundamentos da Matemática. Teoria dos Tipos. Lógica.

ABSTRACT

This work develops a study of a sequence of formal systems, with the objective of investigating the relationship between computation and mathematics, and presents a possible new direction for the development of new proof assistants. The starting point is the Simply Typed Lambda Calculus, and successive extensions are presented, culminating with CoqMT. This work deals only with some aspects of each system, such as the mathematics they capture, and how computation manifests in those systems. Although there are independent discussions of each of those systems elsewhere, here there is an emphasis in the view of the systems as a proper sequence, which enables a clear discussion of recurring ideas. Based on those ideas, this dissertation presents what we called "inductive structures", a generalization of concepts found in the Calculus of Inductive Constructions. This dissertation concludes with examples that illustrate the usage of computation in the presence of such structures, and outlines an implementation based on pre-existing works in the literature.

Keywords: Proof Assistants. Computation. Foundations of Mathematics. Type Theory. Logic.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A ciência da computação possui forte ligação com a lógica, a área da matemática responsável por tratar da prova de teoremas em toda sua generalidade e formalidade. Como fruto moderno dessa ligação, destacamos em particular os assistentes de provas, programas de computador que disponibilizam a seus usuários diversas ferramentas para o auxílio na confecção de provas e teoremas, desde a verificação sintática de afirmações, a buscas automáticas por demonstrações de teoremas. No entanto, poucas são as áreas da matemática que fazem uso destes assistentes. Na matemática, o computador ainda é em grande parte usado apenas pela sua capacidade de calcular. Tais assistentes de prova hoje encontram público na comunidade de cientistas da computação que precisam de maior confiança em seus programas.

Nesse contexto, a questão natural que se coloca é: O que falta para que os assistentes de prova se tornem mais atraentes para os matemáticos? Ou, em outras palavras, de que maneira os recursos computacionais podem ser colocados a serviço da construção de provas matemáticas?

O exemplo a seguir nos dará algumas primeiras ideias a respeito da natureza da questão. Considere um teorema simples como a comutatividade da soma entre números naturais. O que ocorre na maioria dos assistentes de prova é que, a demonstração deste teorema não altera a maneira como o sistema “vê” ou "entende"a soma de dois números. Isto é, o teorema é apenas uma nova afirmativa adicionada ao conjunto de afirmativas válidas. E essa afirmativa, em princípio, deve ser aplicada manualmente quando necessária. Os cientistas da computação em geral estão acostumados com essa natureza explícita dos fatos, e tem o costume de desenvolver técnicas de programação para fazer do explícito um poderoso aliado (e.g. sistemas de tipos fornecendo informação para sugestão de código). Por outro lado, o matemático está habituado a trabalhar com interlocutores com um certo nível de inteligência, e fica frustrado com a repetição. Essa observação aponta para uma deficiência do computador enquanto um assistente do matemático. O computador não aprende um novo conceito simplesmente porque nele foi inserido uma formulação deste conceito. É necessário também programar como o conceito deve ser utilizado, em que condições ele pode ser aplicado, e assim por diante. Em outras palavras, é preciso programar a aplicação automática do conceito pelo computador. Ou seja, a chave para a questão acima é a noção de automação.

Na realidade, parte da repetição que frustra o matemático está associada a uma certa função. Se a prova de um teorema é escrita de forma completa, até os mínimos detalhes, ela pode ser verificada para que se obtenha segurança absoluta com relação à sua corretude. Essa é uma

das grandes vantagens de fazer matemática no computador, pois ele pode realizar esta verificação. O problema é que uma prova deste tipo possui uma grande quantidade de argumentos sobre fatos triviais (muitas vezes repetitivos). Poincaré (1902 apud BARENDREGT; COHEN, 2001, p. 14) aponta uma solução deste problema na seguinte observação: "se a prova de $2+2=4$ é necessária num argumento matemático, então esta parte não é uma prova no senso estrito da palavra, mas uma 'mera verificação'". O que Poincaré parece estar indicando aqui, é que certos passos de uma prova podem ter caráter mecânico ("mera verificação"), em oposição a outros, que tem caráter lógico. E, na medida em que certos passos são mecânicos, eles podem ser omitidos, pois o computador pode realizá-los de maneira automática.

Neste ponto, nós acabamos de encontrar a noção específica de automação que será utilizada nesse trabalho: a capacidade do computador de verificar a validade de certos fatos e certas passagens, sem a necessidade de apresentar uma prova para eles. Note que essa noção de automação responde aos anseios do matemático de duas maneiras. Primeiro, no momento em que ele identifica um passo trivial do argumento, ele pode simplesmente indicar que a prova segue por um raciocínio mecânico (dizendo algo como "é fácil ver que..."). Segundo, quando ele raciocina em nível mais alto, por exemplo, invocando um teorema sem apresentar os detalhes da sua aplicação. Nesse sentido, o caminho natural seria explorar ao máximo essa noção de automação. E, a questão que se coloca então é: Como identificar as partes de uma prova que possuem caráter mecânico, e podem ser omitidas, e as partes que possuem caráter lógico, e precisam ser feitas de maneira manual pelo matemático? Infelizmente, não há uma resposta simples para esta questão. Mesmo um exemplo trivial como $2+2=4$ pode requerer argumentação lógica, na medida em que os fatos aritméticos relevantes ainda não estejam organizados na forma de regras de computação. Por outro lado, um enunciado matemático de uma teoria relativamente sofisticada pode ganhar a aparência mecânica, na medida em que se alcance um compreensão clara e extensiva dos raciocínios que são realizados naquela teoria (e.g. algoritmos de decisão). Ou seja, o nosso entendimento a respeito da matemática é construído a medida que nós fazemos matemática, alterando a percepção que temos da natureza das nossas provas.

Na prática, os pesquisadores da área encontram diversas maneiras de introduzir a capacidade computacional nos assistentes de prova. O lugar mais fundamental aonde a automação é introduzida no sistema consiste na linguagem lógica básica do assistente de prova. Por exemplo, Mizar (MATUSZEWSKI; RUDNICKI, 2005) é um assistente que utiliza como linguagem básica a teoria clássica de primeira ordem dos conjuntos. A única automação oferecida nesta linguagem

é a automação encontrada em qualquer assistente de prova, isto é, a verificação da corretude de aplicação das regras associadas a tal linguagem. Um exemplo mais interessante é dado pelo assistente de prova Coq (THE COQ DEVELOPMENT TEAM, 2017), que tem como linguagem base a teoria dos tipos. A automação adicional que se encontra nessa linguagem é a capacidade que o sistema tem de verificar a identidade entre termos em princípio sintaticamente diferentes. Ainda outro exemplo é dado pelo assistente de prova ACL2 (KAUFMANN; MOORE, 1997). Este assistente é baseado em uma linguagem lógica mais simples (lógica de primeira ordem), que limita as suas aplicações, mas permite uso mais poderoso de algoritmos de semi-decisão. A automação que pode ser encontrada na linguagem básica de um assistente de prova é necessariamente limitada, pois, como característica de projeto, essas linguagens tendem a ser o mais simples possível para facilitar a certificação da corretude do núcleo lógico do sistema. Para lidar com essa simplicidade extrema, os assistentes de prova tipicamente oferecem uma linguagem intermediária para facilitar a construção da prova. As linguagens intermediarias mais populares são as táticas. Informalmente, táticas são pequenos programas, que constroem uma prova a partir da combinação de provas mais simples. Em sistemas como o Coq, a garantia desse processo é baseada no fato de que o objeto de prova construído pela tática deve ser verificado pelo sistema de tipos. Em outros sistemas, como o Isabelle (MULLIGAN), 2011), a linguagem intermediária garante a corretude da aplicação das táticas a cada passo. Uma outra abordagem para obter automação em nível intermediário é encontrada no assistente de prova ACL2, que permite utilizar teoremas já demonstrados como regra de reescrita. Finalmente, existe toda uma variedade de maneiras ad-hoc para estender a capacidade computacional de um assistente de prova. Por exemplo, em seu livro CPDT, Chlipala (CHLIPALA, 2013) apresenta uma tática chamada "crush", que essencialmente realiza uma busca utilizando resultados que já foram demonstrados, para tentar encontrar uma prova para o teorema dado. Um método um pouco mais fundamental consiste na técnica de reflexão, que explora a computação presente na linguagem lógica básica, operando diretamente sobre uma representação da estrutura sintática de termos. Esta técnica é tipicamente encontrada no modo "larga escala", com exemplos como "omega"(PUGH, 1991) e "ring"(BOUTIN, 1997). Mas há também a reflexão em "pequena escala", dada pelo sistema SSreflect (GONTHIER *et al.*, 2016), presente em demonstrações de teoremas notáveis, como o "Odd Order Theorem"(GONTHIER *et al.*, 2013).

Depois de toda essa discussão, nós estamos prontos para dar uma forma mais precisa à questão fundamental que norteia este trabalho. Nós já sabemos que a chave para tornar os

assistentes de prova mais atraentes para os matemáticos é a automação. E nós vimos que essa automação, na realidade, surge como resultado de um entendimento mais claro que é alcançado após a exploração de uma teoria matemática qualquer. Mais precisamente, o conhecimento que é obtido com uma exploração inicial da teoria pode ser reorganizado na forma de regras de computação e algoritmos, que por sua vez dão suporte a novas explorações. Por outro lado, este aspecto gradual e construtivo da automação em geral não aparece de maneira clara nas técnicas que foram revisadas acima. Dessa maneira, nos parece que a área de provas assistidas por computador carece de uma melhor compreensão sobre a maneira como a automação contribui para o processo de criação matemático. De fato, a situação ainda é um pouco mais delicada do que isso, pois não parece existir sequer a percepção de que existe essa carência. Nesse sentido, uma das contribuições dessa dissertação consiste precisamente em formular o problema da automação em assistentes de prova colocando em foco o seu caráter gradual e construtivo. Em outras palavras, a questão de interesse é a seguinte: Será que é possível definir um mecanismo que permita expandir a capacidade computacional de um assistente de provas, onde essa expansão é obtida por meio da reorganização do conhecimento matemático que vai sendo desenvolvido?

Como se pode ver, este é um problema nada trivial. Mesmo a discussão de possíveis soluções requer um conhecimento técnico extensivo sobre como a automação tem sido introduzida nos assistentes de provas atuais. Nesse sentido, a maior parte do corpo deste trabalho consiste na revisão de uma sequência de sistemas formais, com o objetivo de observar o crescimento da matemática que pode ser expressa nesses sistemas, e o crescimento da automação embutida neles. Abaixo, nos descrevemos de maneira sucinta o conteúdo dos próximos capítulos.

No segundo capítulo, nós apresentamos o cálculo lambda simplesmente tipado. Este cálculo é a base da sequência que estudamos. Apesar de inadequado como linguagem lógica básica para um assistente de prova (pois limita-se à lógica proposicional), ele ilustra perfeitamente como um sistema formal pode ser formulado com noções de computação embutidas.

No terceiro capítulo, nós apresentamos o cálculo das construções. É um sistema avançado, que pode ser entendido como a combinação de várias linhas de pesquisa num só ponto. A discussão da interação entre computação e lógica de alta ordem, presentes nesse sistema, é de interesse para toda a dissertação.

No quarto capítulo, nós apresentamos o cálculo das construções indutivas. O assistente de provas Coq (THE COQ DEVELOPMENT TEAM, 2017) utiliza este sistema como base. Nós fazemos uma discussão rica em exemplos sobre o poder da indução, especialmente quando

aliada à computação.

No quinto capítulo, nós apresentamos o CoqMT (STRUB, 2010). Aqui, chegamos ao fim da sequência de sistemas formais. Esta última expansão se destaca pela complexidade envolvida no encaixe entre a nova forma de computação introduzida, e a computação originalmente presente, necessitando uma apresentação focada nos detalhes técnicos.

No sexto e último capítulo, nós apresentamos as nossas conclusões. Em um certo sentido, este é o capítulo mais importante do trabalho, pois ele se propõe a formular uma resposta para a pergunta que deixamos acima: como introduzir automação em um assistente de prova, reorganizando o conteúdo matemático que já foi demonstrado. Ao revisitar o conteúdo dos capítulos anteriores, nós classificamos as diversas tentativas de introdução de automação que foram apresentadas como casos bem-sucedidos ou casos mal-sucedidos. Essa análise eventualmente revela elementos que parecem fornecer uma chave para o nosso problema. Mais especificamente, fica claro que é natural pensar a noção de automação em conjunto com algum tipo de estrutura indutiva envolvendo os termos e fórmulas de uma teoria. De fato, o cálculo das construções indutivas oferece uma maneira padrão de combinar automação e indução, utilizando as noções de redução, eliminadores, construtores, etc. Essa solução, no entanto, exige que a estrutura indutiva seja identificada *a priori* e seja materializada na definição dos termos e funções básicas da teoria. A nossa observação fundamental é que as estruturas indutivas podem ser identificadas durante o processo de desenvolvimento da teoria. E, na medida em que essa identificação tenha sido feita, os teoremas que já foram demonstrados podem ser interpretados como regras de redução, possibilitando um aumento da automação presente no sistema. Essa é a maneira como o conteúdo matemático é reorganizado na forma de regras computacionais.

2 CÁLCULO LAMBDA SIMPLESMENTE TIPADO

Quando fazemos matemática no papel, normalmente seguimos as regras ditadas pelo sistema sobre o qual estamos fazendo matemática. Mas também se ensina que toda a matemática é a princípio redutível para a chamada Teoria dos Conjuntos (ou outros formalismos, como a Teoria das Categorias); de acordo com isso, estaríamos apenas seguindo as regras dessa teoria, e as regras inicialmente consideradas seriam apenas derivadas destas últimas. Na prática, poucos fazem tal redução, devido à um aumento considerável na quantidade de detalhes. No computador, por outro lado, nós temos um agente capaz de gerenciar tais detalhes, o que torna viável o uso prático de um formalismo primitivo a partir do qual possamos expressar todo o restante da matemática. Ao mesmo tempo, nós precisamos de formalismos para trabalharmos com matemática no computador; precisamos representar o jogo matemático - teoremas, definições, provas, entre outros, e como esses conceitos se encaixam - e num computador isto é feito através de linguagens formais.

O formalismo que nós utilizaremos para fazer matemática no computador é um tanto peculiar: a chamada teoria dos tipos. Este formalismo é baseado em duas noções primitivas: *tipos* e *termos*. A ideia é que cada teorema irá corresponder a um tipo, e suas provas vão corresponder a termos deste tipo. Mas, nem todo tipo é um teorema. De fato, tipos correspondem a fórmulas, que podem ser verdadeiras ou falsas. Uma fórmula falsa, é claro, não possui provas, logo o tipo correspondente não possui termos (dizemos neste caso que o tipo não é habitado). Portanto, a atividade de fazer matemática - isto é, demonstrar que certas fórmula são teoremas - utilizando teoria dos tipos consiste em tentar construir termos que habitam o tipos que corresponde ao teorema.

Os *tipos simples* capturam um aspecto essencial do conceito de função: o fato de que uma função possui um argumento que pertence a um domínio bem-definido e retorna um resultado que pertence a um contradomínio bem-definido. Por exemplo, quando escrevemos $f : \mathbb{N} \to B$ nós estamos especificando que a função f tem domínio $\mathbb{N}$ e contradomínio B. Na linguagem dos tipos simples, nós dizemos que a função f tem tipo $\mathbb{N} \to B$, o tipo função formado a partir dos tipos (atômicos) $\mathbb{N}$ e B. Mas o que torna a noção de tipos simples interessante é que tanto o domínio quanto o contradomínio podem ser tipos função. Por exemplo, $A \to (B \to C)$ é o tipo das funções recebem argumentos de tipo A e retornam resultados de tipo $B \to C$.

Os *termos lambda*, por sua vez, capturam outro aspecto essencial do conceito de função: o fato de que elementos do domínio são mapeados em elementos do contradomínio. Por

exemplo, quando escrevemos $f(x) = 2x + 1$, nós estamos especificando como obter o resultado da função f a partir do argumento x. Na linguagem dos termos lambda, o nome da função não é importante, e utiliza-se a notação $\lambda x.(2x+1)$ para especificar a função, destacando apenas o argumento da função e o seu corpo, que indica como calcular o resultado. É interessante notar que a notação lambda é rica o suficiente para definir funções que recebem e/ou retornam outras funções. Por exemplo, $\lambda x.\,(\lambda y.\,(2x+y))$ é a função que, ao receber um certo número a como entrada, retorna a função $\lambda y.\,(2a+y)$. Nessa mesma linguagem, também temos uma notação diferente para a chamada de funções, da forma $(f\,x)$, onde f é a função, e x é o argumento a ser passado para a função. O exemplo anterior pode então ser descrito como a chamada $((\lambda x.\,(\lambda y.\,(2x+y)))\;a)$ retornando $\lambda y.\,(2a+y)$[1].

O *cálculo lambda simplesmente tipado* (ou STLC, onde a sigla é formada a partir da tradução para o inglês) combina os tipos simples com os termos lambda acrescentando regras para associar tipos aos termos. O primeiro passo consiste em associar tipos aos argumentos das funções. Por exemplo, na função $f(x) = 2x + 1$, x tem tipo $\mathbb{N}$ (denotado por $x : \mathbb{N}$), e o termo lambda correspondente passa a ser escrito como $\lambda x : \mathbb{N}.(2x+1)$. Uma vez que os tipos dos argumentos estão bem-definidos, nós podemos associar tipos aos termos lambda. Por exemplo, $\lambda x : A.\,(\lambda y : B.x)$ é a função que recebe uma entrada a do tipo A e retorna uma função do tipo $B \to A$ (a função que sempre retorna a, independente de entrada), e portanto tem tipo $A \to (B \to A)$.

2.1 Aplicações do STLC

O STLC é a teoria dos tipos que captura o fragmento implicacional da lógica proposicional. Nessa correspondência, os tipos simples podem ser interpretados imediatamente como as fórmulas da lógica proposicional, e os tipos habitados correspondem exatamente às formulas verdadeiras. Em particular, se um tipo é habitado sem que se faça qualquer suposição adicional, tal tipo corresponde a uma tautologia. A seguir, veremos alguns exemplos que ilustram a correspondência (mais adiante apresentaremos este resultado de maneira formal):

a) a fórmula proposicional $A \to A$ é claramente uma tautologia (ela é verdadeira independente do valor verdade de A), e isso é comprovado pelo fato de que nós podemos escrever o termo $\lambda x : A.\,(x)$, que é um habitante do tipo $A \to A$

[1] O termo $\lambda x.\,(\lambda y.\,(2x+y))$ também pode ser visto como uma função de dois argumentos; algumas vezes usaremos faremos uso da notação $(F\,a\,b)$ para representar uma chamada de função passando dois valores, o que portanto é na verdade $((F\,a)\;b)$.

(independente de quem é o tipo A, seja ele habitado ou não) [2];

b) por outro lado, a fórmula $A \to B$ é uma contingência. Quando A é verdadeiro e B é falso, o que corresponde à situação em que temos um tipo A com ao menos um habitante (a, digamos) e um tipo B sem habitantes, não existe função de tipo $A \to B$, pois não é possível encontrar um elemento em B para o qual a é mapeado[3]. Por outro lado, quando B é verdadeiro, nós temos um tipo B com ao menos um habitante (b, digamos), e nós podemos apresentar a função $\lambda x : A.b$ do tipo $A \to B$, independente de quem é o tipo A;

c) na discussão acima, nós vimos que o tipo $A \to (B \to A)$ tem o habitante $\lambda x : A.(\lambda y : B.x)$, o que corresponde ao fato de que a fórmula $A \to (B \to A)$ é uma tautologia;

d) finalmente, é importante corrigir uma pequena imprecisão que cometemos há pouco: O STLC captura apenas o fragmento implicacional da lógica proposicional *intuicionista*. O contra-exemplo clássico é a lei de Peirce,$((A \to B) \to A) \to A$, uma tautologia da lógica proposicional que não pode ser provada na lógica intuicionista.

2.2 Definição formal do STLC

A seguir, nós vamos apresentar a definição formal do STLC, que é composta por três elementos: as regras de construção dos tipos simples, as regras de construção dos termos lambda, e o cálculo que define a relação de tipagem.

Definição (STLC)**.** *Seja U um conjunto qualquer, cujos elementos serão considerados como os nossos tipos atômicos. Os* tipos simples τ *do STLC são definidos pela seguinte gramática:*

$\tau ::= \tau \to \tau \mid T \text{ onde } T \in U$

A seguir, seja C um conjunto de constantes, usado para popular tipos atômicos, e seja V um conjunto de variáveis. Os termos lambda t *do STLC são definidos pela seguinte gramática:*

$t ::= x \mid c \mid t\,t \mid \lambda x : \tau.t$

[2] Note que, no caso em que A não é habitado, a função $\lambda x : A.(x)$ ainda é bem-definida, pois obedece à todas as condições exigidas para ser uma função. Por exemplo, mapeia todos os elementos do domínio em algum elemento do contradomínio; a função satisfaz esta condição vacuosamente.

[3] Isto decorre da totalidade das funçoes aqui consideradas

onde $x \in V$ e $c \in C$. Existem, portanto, quatro formas de construir um termo lambda: (1) a partir de uma variável, (2) a partir de uma constante, (3) a partir da aplicação de um termo a outro (onde o primeiro faz o papel de função e o segundo faz o papel do valor passado para esta função), e (4) a partir da abstração lambda, que constrói uma função dados uma variável com seu tipo (que fazem o papel de argumento tipado da função) e um termo (que faz o papel de corpo da função).

Finalmente, nós temos o cálculo que define a relação de tipagem, que associa tipos simples a termos lambda em um contexto de tipagem. Um contexto de tipagem Γ *é um coleção de atribuições da forma $x : \tau$. Considere também um mapeamento fixo que associa a cada $c \in C$ um tipo T_c. A* relação de tipagem $\Gamma \vdash t : \tau$ *(que lemos "o termo t é bem-tipado com tipo τ no contexto Γ") é a menor relação contendo as triplas deriváveis a partir do sistema abaixo.*

$$\frac{x \in V}{\Gamma \cup \{x : \tau\} \vdash x : \tau}\ (1) \qquad \frac{c \in C}{\Gamma \vdash c : T_c}\ (2)$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash e : \sigma}{\Gamma \vdash (\lambda x : \tau . e) : \tau \to \sigma}\ (3) \qquad \frac{\Gamma \vdash t_1 : \sigma \to \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash (t_1\, t_2) : \tau}\ (4)$$

Quando o contexto é vazio, nós utilizamos a notação mais sucinta $\vdash t : \tau$. Dizemos que um termo t é bem-tipado *quando existe um tipo τ tal que $\vdash t : \tau$.*

Para ilustrar o uso do cálculo, nós vamos apresentar mais dois exemplos:

a) uma derivação da tripla $\vdash \lambda f : A \to B . (\lambda a : A . (f\, a)) : (A \to B) \to (A \to B)$, dada pela árvore de derivação a seguir:

$$\dfrac{\dfrac{\dfrac{\dfrac{f \in V}{\{f : A \to B, a : A\} \vdash f : A \to B}\ (1) \qquad \dfrac{a \in V}{\{f : A \to B, a : A\} \vdash a : A}\ (1)}{\{f : A \to B, a : A\} \vdash f\, a : B}\ (4)}{\{f : A \to B\} \vdash \lambda a : A . (f\, a) : A \to B}\ (3)}{\vdash \lambda f : A \to B . (\lambda a : A . (f\, a)) : (A \to B) \to (A \to B)}\ (3)$$

b) como outro exemplo, temos um termo ao qual não conseguimos associar um tipo no contexto vazio: $(\lambda x : A . x)\ (\lambda x : A . x)$ é formado a partir da aplicação do termo $(\lambda x : A . x)$, que deve ser aplicado a termos do tipo A mas está sendo aplicado ao termo $(\lambda x : A . x)$, de tipo $A \to A$. A única regra do cálculo aplicável a uma aplicação, a regra (4), exige que o tipo de um argumento e o tipo do domínio da função a ser aplicada sejam os mesmos, o que torna impossível tipar $(\lambda x : A . x)\ (\lambda x : A . x)$.

> Nós observamos, portanto, que o conjunto dos termos consiste inclusive de objetos sem sentido, e que um dos propósitos da relação de tipagem é estabelecer quais dos termos são valores sensatos, e quais não são.

2.3 Computação no cálculo lambda

Uma vez que fizemos uma boa explicação do STLC enquanto sistema formal, precisamos também dar atenção ao aspecto que o distingue de sistemas formais de similar expressividade: a possibilidade de *computar* com seus termos. Há apenas um único mecanismo básico, a β-redução[4], que pode ser descrita pelo seguinte par de equações:

$(\lambda x : \tau.(b))\ a \to_\beta b\,[x := a]$

Se $t \to_\beta u$ então $v\,t \to_\beta v\,u$, $t\,v \to_\beta u\,v$ e $\lambda x : \tau.t \to_\beta \lambda x : \tau.u$

onde o expressão à direita de $\to_\beta$ na primeira equação indica o termo que é obtido quando substituímos as ocorrências de x em b por a. Esta regra de redução formaliza no sistema a noção operacional associada a aplicação. Por exemplo, a função $\lambda x : \mathbb{N}.(2x+1)$ aplicada a 3, formando o termo $(\lambda x : \mathbb{N}.(2x+1))\ 3$, β-reduz para $(2\cdot 3+1)$[5]. A segunda equação é chamada de regra de compatibilidade, servindo para garantir que seja possível aplicar a β-redução à subtermos. Este é um mecanismo surpreendentemente poderoso; no cálculo lambda sem tipos, é suficiente para tornar o sistema Turing-completo! Mas num cálculo com tipos, como o STLC, a β-redução é mais comportada, e a repetição de sua aplicação sempre chega ao fim. Ou seja, a relação $\to_\beta^*$, o fecho transitivo-reflexivo de $\to_\beta$, é decidível. Outro elemento derivado de $\to_\beta$ do qual faremos uso é a β-equivalência, denotada por $a =_\beta b$, que significa simplesmente $\left(a \to_\beta^* b\right) \wedge \left(b \to_\beta^* a\right)$. E além de ser uma relação que se presta à múltiplas definições, nós temos também que ela é bem-comportada com respeito aos tipos dos termos envolvidos. Em particular, se temos $a_1 : A$ e $a_1 \to_\beta a_2$, então $a_2 : A$. Esta propriedade é chamada de *subject reduction*, e servirá como base para regras importantes em cálculos futuros.

[4] Textos que tratam do cálculo lambda, especialmente o sem tipos, costumam também mencionar a α-redução, usada para lidar com um problema técnico envolvendo nomes das variáveis lambda, o sombreamento. É possível também tratar do problema de nomes das variáveis usando outros artifícios, como índices "de Brujin". Como esses detalhes não são do interesse do trabalho, nós não iremos tratar deles.

[5] O leitor certamente tem em mente que este termo continua a reduzir até uma forma final, 7. Essas reduções são independentes das que consideramos no STLC. Não iremos formalizar como adicionar novas reduções ao STLC, mas fica claro que isto é uma possibilidade, e que torna possível estudar programação sobre diversos domínios usando apenas "ferramentas da lógica proposicional" como estruturas de programação. Pelo isomorfismo de Curry-Howard, a ser tratado mais adiante, podemos obter uma variedade de mecanismos de controle; podemos até dizer que o paradigma resultante contém muitos dos elementos do paradigma da "Programação Funcional" tal como presente em linguagens como Haskell e ML.

Esta relação como um todo ganhará destaque apenas nas próximas seções. Sua decidibilidade significa que é amenável a tratamento por computadores, e usaremos disto para introduzir automação aos próximos sistemas formais. Não podemos fazê-lo agora porque a β-redução não tem nenhuma conexão direta com as regras de tipagem do STLC, mas a apresentação da β-redução é mais clara neste momento, ficando como responsabilidade do leitor retornar a esta seção se dúvidas surgirem sobre a β-redução no futuro.

2.4 Limitações do STLC

Como indicado anteriormente, há uma correspondência entre o STLC e a lógica proposicional. Isto significa que lógica proposicional é a matemática que podemos fazer usando o STLC. Mais precisamente, vale o chamado *isomorfismo de Curry-Howard*: O STLC e o cálculo de dedução natural para a fragmento implicacional da lógica proposicional intuitonista são essencialmente o mesmo sistema formal.

A vantagem deste tipo de resultado é que quaisquer ideias desenvolvidas em um sistema podem ser aplicadas ao outro, mediante releitura com base no isomorfismo. Enquanto que o STLC e suas aplicações são menos conhecidas, da lógica proposicional podemos transportar várias ideias frutíferas. De fato, na apresentação do cálculo, nós utilizamos como modelo o sistema de *dedução natural*, um importante sistema dedutivo desenvolvido para uso em lógica durante o século XX. Nossos exemplos iniciais foram em sua maior parte obtidos da literatura da lógica proposicional.

Na verdade, não é preciso nos limitarmos à lógica implicacional intuitonista. Nós podemos utilizar o isomorfismo para ir além. Para capturar o fragmento implicacional da lógica clássica, basta introduzir a lei de Peirce como axioma ao sistema. Para introduzir uma proposição A arbitrária como um axioma, basta assumir que existe uma constante c que habita o tipo A. No caso da lei de Peirce, para cada par de tipo simples α, β, nós designamos uma constante $c_{\alpha,\beta}$ como habitante do tipo $((\alpha \to \beta) \to \alpha) \to \alpha$.

Além disso, também é possível ir além do fragmento implicacional utilizando algumas ideias simples. Por exemplo, para introduzir a proposição $\bot$, que é sempre falsa e a partir da qual tudo pode ser derivado, nós podemos distinguir um tipo atômico $\bot$ para fazer o mesmo papel, e introduzir axiomas da forma $\bot \to \tau$, para cada tipo simples τ. Uma vez que temos o tipo $\bot$, nós podemos definir a negação de um tipo τ como o tipo função $\tau \to \bot$ (note que este tipo é habitado se e somente se τ não é habitado). Outras ideias simples permitem trabalhar com

os conectivos $\wedge$ e $\vee$ no sistema.

Infelizmente, sendo capaz de capturar apenas a lógica proposicional, o STLC não é apropriado para nossos propósitos. Para que possamos fazer matemática, nós precisamos de um sistema formal que capture pelo menos a lógica de primeira ordem - a Teoria dos Conjuntos é tipicamente feita nesta lógica - e tal lógica requer estruturas capazes de expressar relacionamentos entre proposições que a lógica proposicional não fornece (isto é, alguma forma de quantificação). A outra limitação já foi mencionada pouco mais acima, e consiste no fato de que, o mecanismo de computação no cálculo não tem nenhuma conexão direta com as regras de tipagem.

3 CÁLCULO DAS CONSTRUÇÕES

Nosso objetivo é obtermos um formalismo que capture os princípios que podemos observar como necessários para a realização de qualquer atividade matemática. Mais que isso, precisamos de um formalismo que dê suporte à introdução de computadores como elementos fundamentais para o uso prático. O STLC não é capaz de atingir qualquer uma das duas metas; a matemática nele realizável resume-se à lógica proposicional, e não há nenhum elemento no sistema formal em que um computador possa agir diretamente.

O *Cálculo das Construções* (COQUAND; HUET, 1988) (CC) é um sistema formal muito mais próximo do ideal descrito acima. O CC captura uma lógica de alta ordem - muita da prática matemática pode ser formalizada em lógica de primeira ordem - e veremos que uma adição ao cálculo dos tipos, a regra de conversão, serve como candidata natural para automação. Além disso, o CC pode ser visto como uma extensão do STLC; nós não entraremos em detalhes sobre qual exatamente é esta relação (mas faremos paralelos ao STLC quando conveniente para fins expositórios), convidando leitores interessados a lerem sobre o cubo lambda (BARENDREGT, 1991).

3.1 Elementos do Cálculo das Construções

Do ponto de vista técnico, o ganho oferecido pelo CC é a possibilidade de manipular tipos da mesma maneira que se manipulam termos do STLC, isto é, a possibilidade de definir funções que recebem ou retornam tipos. Obtemos este ganho introduzindo novos elementos, descritos a seguir, através da adição de termos e novas regras ao cálculo.

O primeiro elemento necessário para que isso seja possível é a noção de tipo dos tipos, denotada por $*$, que fará o papel de domínio ou contradomínio das funções que envolvem tipos. Por exemplo, $\lambda x : *.x$ é a função identidade sobre o tipo dos tipos (i.e., ela recebe um tipo como entrada e retorna o mesmo tipo como saída), $\lambda x : \tau.\mathbb{N}$ é uma função constante (i.e., ela recebe um termo de algum tipo τ e retorna o tipo $\mathbb{N}$), e $\lambda C : *.(\lambda x : C.x)$ é a função identidade polimórfica (i.e., ela recebe um tipo e retorna a função identidade sobre o tipo recebido).

O segundo elemento é uma generalização da operação de construção de tipos função, que permite expressar uma dependência entre o termo recebido como entrada e o tipo do termo de saída. A forma geral deste operador é $\Pi x : A.B$. Quando o termo B não possui nenhuma ocorrência da variável x, essa expressão corresponde ao tipo função usual $A \to B$. O caso geral

de uso do operador Π se mostra necessário quando queremos determinar, por exemplo, o tipo da função identidade polimórfica $(\lambda C : *. (\lambda x : C.x))$. Veja que nessa função o tipo do valor de saída depende do termo recebido como entrada, o que nos obriga a utilizar o operador Π para definir o tipo da expressão como $\Pi C : *. (C \to C)$.

No CC, temos que algumas expressões complexas, como $(\lambda x : *.x)\ \mathbb{N}$, são tipos. Podemos observar isto com auxílio do cálculo; como $(\lambda x : *.x)\ \mathbb{N}$ tem tipo $*$, é portanto um tipo (de fato, a expressão β-reduz para $\mathbb{N}$, um tipo conhecido). Precisamos do cálculo para determinar isto porque, como tipos agora podem incluir termos arbitrários, verificar que uma expressão corresponde a um tipo pode depender de um certo termo ser bem-formado. Esta situação difere do que ocorre no STLC, onde não precisávamos do cálculo porque para determinar se um tipo é bem-formado basta observar se ele se conforma à gramática dos tipos simples.

Por razões técnicas, somos compelidos a introduzir outro elemento. Similarmente ao que foi discutido no parágrafo anterior, precisamos lançar mão do cálculo para validar expressões como $* \to *$, e $\Pi C : *. (C \to C)$. Estas expressões também denotam tipos, mas como são tipos função cujo domínio é $*$, não podemos dar a elas tipo $*$ no CC, afim de evitar paradoxos conhecidos da literatura. Estas expressões tem tipo $\Box$, o tipo dos *kinds*, que é o nome encontrado na literatura para o $*$ e estas expressões formadas a partir do $*$. Estes são uma espécie de "tipo de alta ordem", também conhecidos por universos, e as tecnicalidades envolvidas não são relevantes para os propósitos desta dissertação.

3.2 Definição formal do Cálculo das Construções

Podemos agora apresentar os termos do Cálculo das Construções, na sua totalidade.

Definição (Cálculo das Construções)**.** *Sejam C um conjunto de constantes e V um conjunto de variáveis. Os termos t do CC são definidos pela seguinte gramática:*

$$t ::= c \mid x \mid (t\,t) \mid \lambda x : t.t \mid * \mid \Box \mid \Pi x : t.t \tag{3.1}$$

onde $c \in C$ e $x \in V$. Note que a coleção de termos definidos pela gramática inclui também os tipos do cálculo. Além disso, observe que as construções de termos constantes, variáveis e aplicações são tais como no STLC. A construção do termo abstração, por outro lado, apresenta uma diferença: como os tipos deste cálculo são também termos, o tipo da variável da abstração é um termo qualquer. Como vimos, o $*$ é o tipo dos tipos, e o $\Box$ é o tipo do asterisco (e certas expressões a partir dele formadas). Finalmente, a construção do tipo função generalizado é

feita pelo operador Π *a partir de uma variável, um termo (que indica o domínio da função), e um outro termo (que indica o contradomínio da função, e que pode depender da variável).*

Assim como no STLC, a relação de tipagem do CC é definida por meio de um cálculo e um mapeamento fixo que associa a cada $c \in C$ *um tipo* T_c*. Em particular,* $* \in C$*, e seu tipo* T_* *é* $\Box$*. Nós usamos* s*,* s_1 *e* s_2 *para denotar* $*$ *ou* $\Box$*. O cálculo é definido pelas regras a seguir:*

$$\frac{x \in V \qquad \Gamma \vdash \tau : s}{\Gamma \cup \{x : \tau\} \vdash x : \tau} \text{ (1)} \qquad \frac{c \in C}{\Gamma \vdash c : T_c} \text{ (2)}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash e : \sigma \qquad \Gamma \vdash (\Pi x : \tau.\sigma) : s}{\Gamma \vdash (\lambda x : \tau.e) : (\Pi x : \tau.\sigma)} \text{ (3)} \qquad \frac{\Gamma \vdash t_1 : \Pi x : \sigma.\tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash (t_1\, t_2) : (\tau[x := t_2])} \text{ (4)}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash \sigma : s \qquad \tau =_\beta \sigma}{\Gamma \vdash t : \sigma} \text{ (5)} \qquad \frac{\Gamma \vdash \sigma : s_1 \qquad \Gamma \cup \{x : \sigma\} \vdash \tau : s_2}{\Gamma \vdash (\Pi x : \sigma.\tau) : s_2} \text{ (6)}$$

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash \tau : s}{\Gamma \cup \{x : \tau\} \vdash t : \sigma} \text{ (7)}$$

Além das regras, a definição de β*-redução também faz parte do CC, de forma praticamente idêntica ao apresentado na seção 2.1, a não ser pela regra de compatibilidade, que deve ser extendida para os termos adicionais do CC.*

Não é difícil ver que as primeiras quatro regras do cálculo são análogas às regras do STLC. A regra (6) caracteriza os tipos função válidos no sistema (isto é, os tipos τ para os quais é possível derivar $\vdash \tau : *$ ou $\vdash \tau : \Box$). A regra (7) é apenas uma regra estrutural, que estabelece que a demonstração de algo continua sendo verdade mesmo quando hipóteses não-relacionadas são adicionadas; é uma regra que se faz necessária quando nós precisamos tornar compatíveis contextos diferentes numa mesma prova.

O elemento mais interessante do cálculo é a regra (5)[1], que nos levará à ideia de automação na construção de provas. O ponto de partida é a observação de que, no CC, nós temos a possibilidade de construir tipos distintos β-equivalentes (por exemplo, o tipo $(\lambda x : *.x)\ \mathbb{N}$ β-reduz para $\mathbb{N}$). Pela discussão que vimos ao final da seção 2.1, nós deveríamos esperar que esses dois tipos sejam essencialmente o mesmo tipo. O papel da regra (5) é justamente internalizar esse fato no cálculo, estabelecendo que os dois tipos têm os mesmos habitantes. Por outro lado, a prova da β-equivalência entre dois termos pode requerer uma série de β-reduções

[1] Embora a β-redução tenha sido definida apenas para os termos do STLC, é imediato que ela pode ser definida para o CC, sem qualquer alteração.

envolvendo termos intermediários. Essa prova não precisa ser apresentada, pois um computador pode fazer esta verificação de maneira automática (a relação de β-equivalência é decidível). Esse mecanismo ofecere a possibilidade de automatizar partes da construção de uma demonstração matemática, realizando passos meramente mecânicos computacionalmente.

3.3 Aplicações do Cálculo das Construções

A seguir, nós vamos ilustrar o ganho de expressividade (e detalhes de uso do cálculo) com alguns exemplos. Esses exemplos, e outros ao longo do trabalho, fazem uso de uma convenção: A equação $A \equiv B$ significa que estamos definindo A como sendo B. Ou seja, todas as ocorrências de A devem ser entendidas como na verdade ocorrências de B.

3.3.1 Lógica de Primeira Ordem

Objetos $P : A \to *$ (isto é, famílias de tipo indexadas por um tipo A) podem ser interpretados como predicados sobre A: para os elementos $x : A$ em que o tipo $(P\,x)$ é habitado, interpreta-se que o predicado é verdadeiro, e para os elementos $x : A$ em que o tipo $(P\,x)$ não é habitado, interpreta-se que o predicado é falso.

Tipos da forma $\Pi x : A.P(x)$, por sua vez, podem ser interpretados como fórmulas universalmente quantificadas $\forall x : A(P(x))$: Quando o tipo é habitado, existe uma função que retorna um habitante de $(P\,x)$ para todo $x : A$, o que implica $P(x)$ é verdadeiro para todo $x : A$, e portanto a fórmula $\forall x : A(P(x))$ é verdadeira. Quando o tipo não é habitado, interpreta-se que a fórmula é falsa. Nesse sentido, estamos interpretando o formador de tipos Π como o quantificador universal intuicionista.

Com esses dois elementos, nós podemos escrever, por exemplo, o teorema da implicação quantificada $\forall x(\phi \to \psi) \to (\forall x(\phi) \to \forall x(\psi))$ em CC como $\Pi x : U.(\phi \to \psi) \to (\Pi x : U.(\phi) \to \Pi x : U.(\psi))$. Utilizando as regras do cálculo, não é difícil verificar que o termo

$$\lambda f : (\Pi x : U.(\phi \to \psi)).(\lambda g : (\Pi x : U.\phi).(\lambda x : U.((f\,x)(g\,x))))$$

é um habitante deste tipo. Outros objetos do CC podem ser interpretados como elementos da

lógica de primeira ordem, da seguinte forma:

$$\begin{aligned}
\bot &\equiv \Pi C : *.C, \\
A \wedge B &\equiv \Pi C : *. (A \to B \to C) \to C, \\
A \vee B &\equiv \Pi C : *. (A \to C) \to (B \to C) \to C, \\
\exists x : A. (P\ x) &\equiv \Pi C : *. (\Pi x : A. ((P\ x) \to C)) \to C
\end{aligned}$$

e conectivos derivados, $\neg P \equiv (P \to \bot)$ e $A \leftrightarrow B \equiv (A \to B) \wedge (B \to A)$. A formação desses objetos é uma técnica elegante, conhecida da literatura, inspirada nas regras de introdução do cálculo de dedução natural para a lógica de primeira ordem e em codificações de dados para o cálculo lambda sem tipos. Nós veremos uma outra aplicação dessa técnica ainda nesses exemplos.

3.3.2 Lógica de Alta Ordem

No exemplos envolvendo lógica de primeira ordem, nós observamos uma correspondência entre o quantificador universal $\forall$ e o construtor de tipos Π. De fato, como nós podemos utilizar Π para quantificar sobre tipos além de atômicos, nós temos uma correspondência com lógicas de alta ordem. Por exemplo, dada uma relação $R : A \to A \to *$, onde temos aRb se e somente se o tipo $R\ a\ b$ é habitado, nós podemos definir uma fórmula $WF_{A,R} : *$ tal que $WF_{A,R}$ é verdadeira se e somente se R é uma relação bem-fundada: $WF_{A,R} \equiv \Pi P : (A \to *). ((\Pi x : A. ((\Pi y : A. ((R\ y\ x) \to (P\ y))) \to (P\ x))) \to (\Pi x : A. (P\ x)))$.

Da mesma forma que a presença da lógica de primeira ordem nos permite definir certos objetos através de enunciados lógicos, nós também podemos usar a lógica de alta ordem para cumprir este papel. Como exemplo estendido, nós vamos definir a igualdade. A base para isso será a lei de Leibniz, também conhecida como a identidade dos indiscerníveis: dois objetos são idênticos se e somente se eles são indiscerníveis, isto é, satisfazem as mesmas propriedades. A noção de indiscernibilidade entre $x : A$ e $y : A$ é capturada pelo seguinte tipo[2]:

$$\Pi P : (A \to *). ((P\,x) \leftrightarrow (P\,y))$$

ou seja, x e y são indiscerníveis se e somente se existe uma função (i.e., um habitante deste tipo) que dado um predicado P, retorna um par de funções $f_P : (P\,x) \to (P\,y)$ e $g_P : (P\,y) \to (P\,x)$.

[2] Note que esse tipo corresponde a um enunciado em lógica de alta ordem, uma vez que há uma quantificação sobre proposições.

As funções f e g testemunham o fato de que a verdade de $P(x)$ implica na verdade de $P(y)$ e vice-versa. Por exemplo, x é indiscernível de x, pois nesse caso basta considerar a função que, ao receber um predicado P, retorna o par de identidades $(id_{(Px)}, id_{(Px)})$.

A definição de igualdade é $a =_A b \equiv \lambda A : *. (\Pi P : (A \to *). ((P\,x) \to (P\,y)))$, mas nós vamos usar simplesmente $a = b$, pois usaremos a igualdade em contextos onde o tipo A será conhecido. O leitor atento perceberá que a ocorrência de $\leftrightarrow$ foi substituída por $\to$. De fato, as duas definições são equivalentes[3].

3.3.3 Aritmética

Nós introduzimos o tipo $\mathbb{N}$, a constante 0, a função sucessor S, e as operações de soma e multiplicação através de axiomas: $\mathbb{N} : *$, $0 : \mathbb{N}$, $S : \mathbb{N} \to \mathbb{N}$, $+ : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$, $\cdot : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. Nós vamos utilizar a notação usual para números, onde $1 \equiv S\,0$, $2 \equiv S\,(S\,0)$, e assim por diante.

Os axiomas da aritmética, na formulação de segunda-ordem, são descritos da seguinte maneira:

$a1 : \Pi m : \mathbb{N}. (((S\,m) = 0) \to \bot)$

$a2 : \Pi m : \mathbb{N}. (\Pi n : \mathbb{N}. (((S\,m) = (S\,n)) \to (m = n)))$

$a3 : \Pi n : \mathbb{N}. ((0 = n) \vee \Sigma m : \mathbb{N}. (S\,m) = n)$

$a4 : \Pi m : \mathbb{N}. (m + 0 = m)$

$a5 : \Pi m : \mathbb{N}. (\Pi n : \mathbb{N}. (m + (S\,n) = (S\,(m + n))))$

$a6 : \Pi m : \mathbb{N}. (m \cdot 0 = 0)$

$a7 : \Pi m : \mathbb{N}. (\Pi n : \mathbb{N}. (m \cdot (S\,n) = (m \cdot n) + m))$

$a8 : \Pi P : \mathbb{N} \to *. ((P\,0) \to (\Pi n : \mathbb{N}. (P\,n) \to (P\,(S\,n))) \to (\Pi n : \mathbb{N}. (P\,n)))$

Com os axiomas listados acima, nós podemos provar, por exemplo, o "teorema" $1 + 1 = 2$. A seguir, nós vamos ver a construção passo a passo do termo que corresponde à prova deste teorema.

a) O primeiro passo da construção consiste em utilizar o axioma $a4$ para obter o termo $(a4\,1)$ do tipo $1 + 0 = 1$. Lembre que, pela definição de igualdade, esse termo é uma função que, dado um predicado P e um termo do tipo $P\,(1 + 0)$, retorna um termo do tipo $P\,1$.

[3] Esta observação requer uma demonstração, que não é do nosso interesse.

b) O próximo passo consiste em definir o predicado que será utilizado na prova (isto é, aplicado a $(a4\,1)$), uma função do tipo $\mathbb{N} \to *$. Nós vamos utilizar a função $F \equiv \lambda n : \mathbb{N}.\,(1+(S\,0) = (S\,n))$.

c) A seguir, nós precisamos de um termo do tipo $F\ (1+0)$. Para isso, nós utilizamos o axioma $a5$, para obter o termo $((a5\,1)\ 0) : (1+(S\,0) = (S\ (1+0)))$. Note que estritamente falando, o termo $((a5\,1)\ 0)$ não é do tipo $F\ (1+0)$. Mas, como $F\ (1+0) =_\beta (1+(S\,0) = (S\ (1+0)))$, a regra de conversão nos permite concluir que $((a5\,1)\ 0) : F\ (1+0)$.

d) Agora, a aplicação da função $(a4\,1)$ aos argumentos F e $((a5\,1)\ 0)$ nos fornece o termo $((a4\,1)\ F)\ ((a5\,1)\ 0) : F\,1$. Para completar a prova, basta observar $F\,1 =_\beta (1+(S\,0) = (S\,1))$, e aplicar a regra de conversão para obtermos finalmente $((a4\,1)\ F)\ ((a5\,1)\ 0) : (1+(S\,0) = (S\,1))$. Note que $(1+(S\,0) = (S\,1))$ é apenas outra maneira de escrever $1+1=2$.

3.3.4 Aritmética com Numerais de Church

Na demonstração anterior, apesar da regra de conversão ter sido utilizada, não se pode dizer que houve automação na construção da prova. Para ver isso, basta notar que a demonstração de, por exemplo, $5+5=10$ envolveria várias instanciações dos axiomas $a4$ e $a5$, e uma série de operações de reescrita. A seguir, nós vamos ver como uma representação mais adequada dos números permite automatizar parte da construção da prova de teoremas sobre números naturais.

O esquema de Church para a representação dos números naturais é:

$$
\begin{aligned}
0 &\equiv \lambda C.\,(\lambda f.\,(\lambda x.x)) \\
1 &\equiv \lambda C.\,(\lambda f.\,(\lambda x.\,(f\,x))) \\
2 &\equiv \lambda C.\,(\lambda f.\,(\lambda x.\,(f\ (f\,x))))
\end{aligned}
$$

e, em geral,

$$
k \equiv \lambda C.\left(\lambda f.\left(\lambda x.\left(\underbrace{f\ (f\ \ldots(f\,x))}_{k \text{ vezes}}\right)\right)\right)
$$

onde as indicações de tipo $C : *$, $f : C \to C$ e $x : C$ foram omitidas por clareza. Os termos acima são conhecidos como *numerais de Church*.

Este esquema representa um número natural n como uma função de alta ordem que itera n vezes a aplicação de uma função f a um argumento x. Nesse sentido, os numerais de Church são habitantes do tipo $\Pi C : *. ((C \to C) \to (C \to C))$. De fato, é possível mostrar que os numerais de Church são os únicos habitantes deste tipo(BÖHM; BERARDUCCI, 1985), que será denotado consequentemente por $\mathbb{N}$.

Uma vez que temos essas definições, a função soma pode ser definida por

$$+ \equiv \lambda n : \mathbb{N}. (\lambda m : \mathbb{N}. (\lambda C. \lambda f. \lambda x. (n C f\ (m C f x))))$$

Não é difícil verificar que essa definição implica que

$$1 + 1 \to_\beta 2$$

Agora, para provar o teorema $1 + 1 = 2$, basta apresentar um habitante do tipo $2 = 2$, pois a regra de conversão implica que esse termo também é um habitante de $1 + 1 = 2$.

A automação na construção desta prova consiste em que a operação de reescrita (que foi necessária na prova baseada em axiomas) foi substituída por uma β-redução, e essa β-redução não precisa ser indicada, pois a regra de conversão é capaz de verificar automaticamente que dois tipos são β-equivalentes. Em outras palavras, parte do trabalho de construção da prova foi delegado para uma computação realizada na verificação da aplicação da regra de conversão.

3.4 Limitações do Cálculo das Construções

Nos exemplos anteriores, nós vimos duas formas de representar a aritmética de números naturais no CC: aritmética com numerais de Church, e aritmética feita com axiomas.

Como vimos na introdução à aritmética com numerais de Church, a aritmética feita com axiomas não é automatizada. Mas há um axioma de tal aritmética que não pode ser construído para numerais de Church, o axioma da indução. A falta deste axioma limita severamente nossa capacidade de raciocínio aritmético.

Muito do que nós sabemos sobre automação aplicada a estruturas matemáticas foi desenvolvido para lidar com dificuldades presentes no estudo de computação. Tais estruturas costumam ser mais simples do que se trata usualmente em matemática (por exemplo, tendem a limitar-se a cardinalidade $\aleph_0$). Portanto, é natural esperar que a automação que desejamos usar seja compatível com este tipo de estrutura.

Infelizmente, o cerne destas estruturas (que incluem o conjunto dos números naturais), o mecanismo de *indução*, não é derivável no CC. Os problemas são descritos de forma

abreviada em (PAULIN-MOHRING, 1993), basicamente[4]. De fato, não apenas há a falha em derivar a regra de indução propriamente dita, mas também problemas como não ser capaz de provar dois termos distintos como sendo tal, ou tipos indutivos que incluem termos além daqueles dados pelos construtores. O cálculo que nós vamos apresentar a seguir oferece uma solução para esses problemas.

[4] O artigo citado trata apenas da codificação impredicativa, mas resultados como (GEUVERS, 2001) mostram uma falha geral em definir a indução, que, pela conservatividade mostrada em (COQUAND, 1989), aplica-se ao CC.

4 CÁLCULO DAS CONSTRUÇÕES INDUTIVAS

O Cálculo das Construções Indutivas (CIC) é uma evolução do CC que permite definir e manipular famílias indutivas. Em sua versão mais simples, uma família indutiva corresponde à noção de tipo indutivo, isto é, um tipo cuja coleção de habitantes possui estrutura indutiva. A seguir, nós vamos explorar esse conceito com a definição indutiva do tipo dos números naturais. A maneira mais simples de definir os números naturais[1] no CIC é:

ind $\quad nat : *. \{Z : nat \mid S : nat \to nat\} .nat\_ind$

Essa expressão introduz o tipo *nat* indicando que Z é um habitante de *nat* e fornecendo uma função S que descreve outros habitantes de *nat*: (SZ), $(S\,(SZ))$, $(S\,(S\,(SZ)))$, etc. Para que tenhamos uma coleção propriamente indutiva, com as propriedades que se esperam de tal (por exemplo, o fato que os únicos habitantes de *nat* são aqueles descritos na frase anterior) a definição é acompanhada por um princípio de indução representado pelo termo *nat_ind* que aparece no final da expressão. O termo *nat_ind* tem seu tipo determinado pela expressão; não entraremos nos detalhes do processo mas veremos qual é o tipo em outra parte desta subseção.

Note que essa definição do tipo dos números naturais é essencialmente a mesma que a definição em CC apresentada no exemplo 2 da seção anterior. A única diferença, até o momento, é que lá a definição foi formulada em termos de uma coleção de axiomas. De fato, o princípio de indução *nat_ind* na definição acima enquadra o axioma $a8$ na definição da seção 3.3.3.

O que diferencia *nat_ind* do axioma é a possibilidade de computar com este termo. Temos isto quando tratamos dos chamados *eliminadores*[2]. O próprio *nat_ind* é um eliminador; em particular, ele é um eliminador *forte* e *dependente*. Veremos primeiro um eliminador *fraco* e *não-dependente*, construído a partir do princípio de indução. Para *nat* tal eliminador tem a seguinte tipo:

$$elim\_nat : \Pi C : *. (C \to ((C \to C) \to (nat \to C)))$$

O eliminador de um tipo indutivo implementa a noção de definição por recursão primitiva. Intuitivamente, a recursão primitiva permite definir funções através da aplicação

[1] Nós iremos trabalhar com a representação unária. Nós poderíamos utilizar, por exemplo, uma representação binária, ou decimal, mas estas não são úteis para fins expositivos.

[2] Nós ainda veremos exatamente qual o papel do eliminador, mas é interessante mencionar que a nomenclatura é baseada em dedução natural, onde cada conectivo é associado a regras de introdução e eliminação.

sucessiva, controlada pelo argumento de entrada, de uma função mais simples sobre um valor inicial fixo. Por exemplo, para definir a função *double* utilizando o esquema de recursão primitiva, a função simples é a composição $(S \circ S)$ e o valor fixo é Z. Formalmente, a construção utilizando o eliminador *elim_nat* é feita da seguinte maneira:

$$double \equiv (elim\_nat\ nat\ Z\ (S \circ S)) \tag{4.1}$$

Na expressão do lado direito, o primeiro argumento, *nat*, informa que estamos construindo uma função que retorna números naturais (que o valor de entrada também é um número natural fica implícito pelo fato de que nós estamos utilizando o eliminador *elim_nat*). Os outros dois argumentos indicam o valor fixo e a função simples, necessários para a definição por recursão primitiva.

Mas, com o que apresentamos sobre o CIC até o momento, a única coisa que podemos dizer a respeito do termo $(elim\_nat\ nat\ Z\ (S \circ S))$ é que ele possui tipo $nat \to nat$. O mecanismo da recursão primitiva, que define o comportamento da função *double*, é capturado no formalismo por uma identificação entre os termos:

$$((elim\_nat\ nat\ Z\ (S \circ S))\ Z) =_\iota Z$$

e entre os termos:

$$((elim\_nat\ nat\ Z\ (S \circ S))\ (S\,x)) =_\iota (S \circ S)\ ((elim\_nat\ nat\ Z\ (S \circ S))\ x)$$

Veja que, de fato, essas identificações são suficientes para capturar o comportamento das definições por recursão primitiva: aplicação sucessiva de uma função simples a um valor fixo. Por exemplo,

$$\begin{aligned}
(double\ (S\ (S\,Z))) &\equiv ((elim_nat\ nat\ Z\ (S \circ S))\ (S\ (S\,Z))) \\
&=_\iota (S \circ S)\ ((elim_nat\ nat\ Z\ (S \circ S))\ (S\,Z)) \\
&=_\iota (S \circ S)\ ((S \circ S)\ ((elim_nat\ nat\ Z\ (S \circ S))\ Z)) \\
&=_\iota (S \circ S)\ ((S \circ S)\ Z) \\
&=_\beta (S\ (S\ (S\ (S\,Z))))
\end{aligned}$$

Note a aplicação sucessiva da função simples $(S \circ S)$ ao valor fixo Z na penúltima linha.

No CIC, as identificações indicadas pelo símbolo $=_\iota$ acima são implementadas por meio de uma nova regra de redução: a ι-redução. Mais precisamente, a ι-redução (denotada por

$\rightarrow_\iota$) para o tipo indutivo *nat* introduzido acima, consiste nas regras de reescrita abaixo:

$$((elim\_nat\ T\ t\ f)\ Z) \rightarrow_\iota \quad t$$
$$((elim\_nat\ T\ t\ f)\ (S\,n)) \rightarrow_\iota \quad f\ ((elim\_nat\ T\ t\ f)\ n)$$

onde T, t e f são termos quaisquer, em conjunto com uma regra de compatibilidade análoga à definida na β-redução. O fecho reflexivo transitivo $\rightarrow_i^*$ e a relação de ι-equivalência $=_i$ são definidos de maneira análoga ao que foi feito para a β-redução.

Complementando a definição do tipo *nat*, o próximo passo consiste em introduzir as funções de soma e multiplicação:

$$\begin{aligned} + &\equiv \lambda n : nat.\,(elim\_nat\ nat\ n\ S) \\ \times &\equiv \lambda n : nat.\,(elim\_nat\ nat\ Z\ (+\,n)) \end{aligned}$$

O exemplo abaixo permite ver como o termo $+$ acima captura a lógica da operação da soma. (o leitor é convidado a fazer um exemplo análogo para a multiplicação)

$$\begin{aligned} (2+2) &\equiv ((\lambda n : nat.\,(elim\_nat\ nat\ n\ S))\ (S\ (S\,Z)))\ (S\ (S\,Z)) \\ &\rightarrow_\beta ((elim\_nat\ nat\ (S\ (S\,Z))\ S))\ (S\ (S\,Z)) \\ &\rightarrow_\iota S\ (((elim\_nat\ nat\ (S\ (S\,Z))\ S))\ (S\,Z)) \\ &\rightarrow_\iota S\ (S\ (((elim\_nat\ nat\ (S\ (S\,Z))\ S))\ Z)) \\ &\rightarrow_\iota (S\ (S\ (S\ (S\,Z)))) \end{aligned}$$

4.1 Tipos indutivos no CIC

A seguir, nós veremos outros exemplos que ilustram características adicionais dos tipos indutivos no CIC.

4.1.1 Tipos Finitos

Os tipos indutivos mais simples que existem são aqueles que possuem um número finito de habitantes. Abaixo, temos tipos que possuem exatamente 0, 1 e 2 habitantes, respectivamente:

ind $\quad void : *.\,\{\}\,.void\_ind$

ind $\quad unit : *.\,\{tt : unit\}\,.unit\_ind$

ind $\quad bool : *.\,\{true : bool \mid false : bool\}\,.bool\_ind$

A ideia geral por trás dos eliminador de tipos I indutivos finitos simples é: Dado um tipo C a habitar, um elemento i de I a eliminar, e um habitante de C para cada construtor de I, o eliminador de I fornece um elemento de C, mais precisamente o elemento associado ao construtor usado para construir i. O eliminador de *unit* é trivial.

O eliminador de *void* é interessante. Sabemos de discussões anteriores que o tipo não habitado corresponde ao $\perp$. O eliminador de *void* corresponde ao chamado "princípio do absurdo"; como não há construtores a serem considerados, o eliminador fornece um elemento (prova, pela correspondência) de C, para qualquer C, bastando apenas que seja fornecido um elemento de *void*.

O eliminador não-dependente de *bool*, $elim\_bool$, nos dá a capacidade de programar as operações usuais sobre os booleanos. Como ilustração, a seguir temos as definições das funções que implementam os operadores booleanos NOT e AND:

$$\begin{aligned} \text{NOT} &\equiv (elim\_bool\ bool\ false\ true) \\ \text{AND} &\equiv \lambda a : bool.(elim\_bool\ bool\ a\ false) \end{aligned}$$

4.1.2 Tipos Agregados

Uma outra possibilidade consiste em construir os habitantes do tipo indutivo a partir dos habitantes de um outro tipo. Para ilustrar essa ideia nós vamos apresentar uma possível definição do tipo dos números inteiros, a partir dos habitantes de *nat*.

$$\begin{aligned} &\text{ind}\quad int : * \\ &\qquad .\{ZI : int \mid PI : nat \to int \mid NI : nat \to int\} \\ &\qquad .int\_ind \end{aligned}$$

Nessa construção, *ZI* representa o número inteiro zero, e *PI* e *NI* produzem os números inteiros positivos e negativos, respectivamente, a partir de números naturais. Por exemplo, os termos $(PI\,Z)$ e $(NI\,(S\,Z))$ são interpretados como os números inteiros $+1$ e -2 respectivamente. Nesse sentido, *PI* e *NI* podem ser vistos como os sinais $+$ e $-$.

Nas definições indutivas, os termos que definem os habitantes do tipo também são conhecidos como *construtores* do tipo indutivo. Por exemplo, na definição acima, *ZI* é um construtor nulário (isto é, não possui argumentos), e portanto constrói apenas um habitante, que é dado precisamente pelo termo *ZI*. Por outro lado, *PI* e *NI* são construtores unários (isto é, possuem um argumento), e constroem um habitante do tipo para cada valor do argumento. É importante destacar que a construção dos habitantes de um tipo indutivo é uma operação sintática,

no sentido de que o termo $(PI\ (SZ))$, por exemplo, não "retorna" um habitante do tipo *int*, mas essa expressão é uma construção sintática que representa um habitante do tipo.

4.1.3 Famílias Indutivas

Como mencionamos no início da seção, o CIC também possui o conceito de famílias indutivas, que permite definir simultaneamente uma coleção de tipos indutivos, indexada por um ou mais tipos. Um exemplo simples desse conceito é a família dos tipos dos pares ordenados:

$$\begin{aligned}&\text{ind}\quad prod : * \to * \to *\\&\qquad .\{pp : \Pi A : *.(\Pi B : *(A \to B \to (prod\,A\,B)))\}\\&\qquad . prod\_ind\end{aligned}$$

Antes de examinar os detalhes dessa definição, é útil observar que essa construção corresponde à noção de produto cartesiano: $(prod\,A\,B)$ é o produto cartesiano dos tipos A e B, habitado por pares ordenados da forma (a,b), onde $a : A$ e $b : B$. Veja que *prod* é uma família de tipos indexada por $*$ e $*$, mais especificamente a família dos tipos $(prod\,A\,B)$ para todo par de tipos A e B.

A primeira parte da construção acima, $prod : * \to * \to *$, indica que nós estamos definindo uma família indexada por dois tipos. A segunda parte é a descrição do único construtor da família, pp. Este construtor recebe quatro argumentos, os tipos A e B dos dois componentes do par, e dois habitantes $a : A$ e $b : B$, e constrói o habitante:

$$(pp\,A\,B\,a\,b) : (prod\,A\,B)$$

Note que o único construtor pp constrói habitantes de todos os tipos da família. No exemplo acima, ele está construindo o habitante $(pp\,A\,B\,a\,b)$ do tipo $(prod\,A\,B)$ da família *prod*.

O eliminador não-dependente *elim_prod* segue o que foi descrito sobre tipos finitos, no primeiro exemplo dessa lista. Ele merece nossa atenção por envolver uma família indutiva, tendo o seguinte tipo:

$$\begin{aligned}&elim\_prod : \Pi P : * \to * \to *.(\\&(\Pi A, B : *.(A \to B \to (P\ A\ B))) \to\\&\Pi A, B : *.((prod\ A\ B) \to (P\ A\ B)))\end{aligned}$$

O eliminador não-dependente começa com ΠP, onde P é a família que o resultado final do eliminador habita. P tem o mesmo tipo de *prod*, uma simetria em linha com o que ocorre

no caso já exemplificado, o do eliminador de um único tipo. Há em seguida termos com os mesmos tipos dos construtores, exceto pela conclusão, que é o tipo do membro da família sendo construído, e finalmente temos uma quantificação sobre todos os parâmetros da família, afim de que o eliminador possa de fato eliminar elementos de qualquer tipo da família.

Na prática, fazemos uso de um eliminador mais simples:

$$elim\_prod\_param : \Pi A, B, P : *. ((A \to B \to P) \to prod\ A\ B \to P)$$

Este eliminador decorre da seguinte observação: os tipos A e B são essencialmente parâmetros invariantes com respeito à definição. Poderíamos imaginar, por exemplo, uma definição de *prod* e termos associados num contexto com A e B livres, e então remover A e B do contexto formando termos lambda; este processo geraria um termo com o tipo de $elim\_prod\_param$ a partir do eliminador não-dependente. De fato, implementações do CIC oferecem recursos para este tipo de técnica. Mas devemos deixar claro que isto não representa qualquer alteração na expressividade da linguagem; é possível derivar $elim\_prod$ a partir de $elim\_prod\_param$, e vice-versa.

4.1.4 Família Indutiva com Recursão

Este entendimento de famílias é suficiente para formarmos a família das palavras parametrizadas sobre alfabetos. Esta é dada pelo seguinte:

$$\begin{aligned} \text{ind}\quad & word : * \to * \\ & .\{empty\_w : \Pi A : *. (word\ A) \mid cons\_w : \Pi A : *. (A \to word\ A \to word\ A)\} \\ & .word\_ind \end{aligned}$$

onde temos que $word\ A$ é o tipo das palavras formadas a partir de um alfabeto A. Os dois construtores são intuitivos: $empty\_w$ constrói a palavra vazia, e $cons\_w$ constrói uma palavra a partir de uma outra palavra e um símbolo a adicionar.

4.1.5 Igualdade definida Indutivamente

Um exemplo ligeiramente mais sofisticado do uso das famílias indutivas é dado pela seguinte definição da noção de igualdade:

$$\text{ind}\, eq : (\Pi A : *. (A \to A \to *)) . \{eqrefl : \Pi A : *. \Pi x : A. (eq\, A\, x\, x)\} . eq\_ind$$

Na primeira parte da construção, o fragmento $eq : (\Pi A : *. (A \to A \to *))$ indica que nós estamos definindo uma família indexada por um tipo A e dois habitantes de A. A ideia aqui é que o tipo

$eq\,A\,a\,b$ é habitado se e somente se os termos $a : A$ e $b : A$ são iguais. Esta ideia é implementada pelo construtor

$$eqrefl : \Pi A : *.\Pi x : A.\,(eq\,A\,x\,x)$$

onde, dado um tipo A e um habitante deste tipo $x : A$, $eqrefl$ constrói o habitante $(eqrefl\,A\,x)$ do tipo $(eq\,A\,x\,x)$, de modo que os tipos habitados na família eq são precisamente os tipos da forma $(eq\,A\,x\,x)$.

É interessante comparar esta definição da igualdade com a definição que foi apresentada na seção anterior para o CC. Como vimos, a definição de igualdade no CC é baseada na lei de Leibniz:

$$\Pi P : (A \to *)\,.\,((P\,x) \leftrightarrow (P\,y))$$

Seguindo o indicado no exemplo de *prod*, obtemos o seguinte eliminador:

$$\begin{aligned} elim\_eq : \Pi P : (\Pi A : *.\,(A \to A \to *))\,.(\\ (\Pi A : *.\Pi x : A.\,(P\ A\ x\ x)) \to \\ \Pi A : *.\Pi x, y : A.\,((eq\ A\ x\ y) \to (P\ A\ x\ y))) \end{aligned}$$

Não é difícil mostrar que a lei de Leibniz pode ser derivada a partir de *elim_eq* (observe que a igualdade tal como definida no CC é um objeto do mesmo tipo que o primeiro argumento). Portanto, é possível operar com a definição do CIC da mesma maneira como é feito no CC[3]. Por outro lado, a definição de igualdade do CIC apresenta algumas vantagens teóricas (por exemplo, é possível demonstrar a unicidade das provas de igualdade para tipos decidíveis no CIC (HEDBERG, 1998)) mas essa discussão escapa do escopo do nosso trabalho.

4.1.6 Tipos Indutivos Funcionais

Podemos representar um subconjunto dos números ordinais, a segunda classe de números, da seguinte forma:

$$\text{ind}\quad ord : *.\,\{ord\_z : ord \mid ord\_s : ord \to ord \mid ord\_lim : (nat \to ord) \to ord\}\,.nat\_ind$$

[3] Assim como no exemplo anterior, é possível obter um eliminador mais simples a partir da observação que parâmetros de eq são invariantes. Neste caso, apenas os dois primeiros parâmetros, dos três de eq, podem ser assim simplificados. Isto resulta em $elim\_eq\_param : \Pi A : *, x : A, P : (A \to *)\,.\,((P\,x) \to \Pi y : A.\,((eq\ A\ x\ y) \to (P\ y)))$, um eliminador a partir do qual é mais fácil ver a conexão com a lei de Leibniz.

Este exemplo ilustra uma forma diferente de utilizar o tipo que está sendo definido, demonstrado no argumento do construtor ord_lim, que é de tipo $nat \to ord$. Este construtor forma um ordinal a partir de uma enumeração de outros ordinais, entendido como o supremo dentre a coleção enumerada. Note que mesmo esse entendimento é algo que também deve ser formalizado, em adição à definição em si.

Podemos utilizar números ordinais em conjunto com computação lançando mão do eliminador $elim\_ord : \Pi P : *. (P \to (P \to P) \to ((nat \to P) \to P) \to ord \to P)$. Considere, por exemplo, um tipo U, e seja P é o tipo dos subconjuntos de U (o que pode ser entendido como $U \to *$, o tipo dos predicados sobre U). Dado um subconjunto inicial (um elemento de P), uma forma de construir um novo subconjunto a partir de outro (um elemento de $P \to P$), e uma forma de construir um novo subconjunto a partir de uma enumeração de subconjuntos (um elemento de $(nat \to P) \to P$, por exemplo, a operação de união), o eliminador $elim\_ord$, dado um ordinal n, fornece a n-ésima iteração dessas duas operações sobre o conjunto inicial.

4.2 Aspectos formais do CIC

A definição formal do Cálculo das Construções Indutivas envolve diversas dificuldades técnicas, e por esse motivo será omitida (o leitor interessado pode consultar (PAULIN-MOHRING, 2015)). Essa definição basicamente estende o CC em três direções: (a) hierarquia de universos de tipos, (b) famílias indutivas, (c) generalização da regra de conversão para incorporar a ι-redução. A hierarquia de universos de tipos pode ser entendida como uma generalização dos tipos $*$ e $\Box$ que vimos no CC, isto é, coleções de tipos de alta ordem. Essa hierarquia serve para evitar paradoxos que aparecem quando se manipula tipos muito grandes sem o devido cuidado(SOZEAU; TABAREAU, 2014). Esse tópico não é muito relevante no contexto do nosso trabalho, e faremos menção a ele quando necessário. A seguir, nós faremos uma breve discussão dos outros dois pontos.

Com relação às famílias indutivas, é preciso em primeiro lugar introduzir regras de boa-formação para definições indutivas. Entretanto, em todos os exemplos de famílias indutivas que veremos nesse trabalho, a verificação de que elas são bem-formadas é imediata. Assim, nós não vamos detalhar tais regras, e vamos apenas encaminhar o leitor interessado para (PAULIN-MOHRING, 1993)), ou para (THE COQ DEVELOPMENT TEAM, 2017).

Além disso, é preciso explicar como as definições indutivas se relacionam com o restante do cálculo. As definições indutivas podem ser entendidas como uma maneira compacta

de introduzir uma coleção de axiomas no sistema. Por exemplo, a definição do tipo indutivo *nat* corresponde aos seguintes axiomas[4]:

- $nat : *$
- $Z : nat$
- $S : nat \to nat$
- $nat\_ind\ :\ \Pi P : nat \to *.\,((P\,0) \to (\Pi n : nat.\,(P\,n) \to (P\,(S\,n))) \to (\Pi n : nat.\,(P\,n)))$

Uma vez que esses axiomas foram registrados no contexto através da declaração do tipo *nat*, essa informação pode ser utilizada para verificar que certos termos estão bem-tipados. Mais precisamente, denotando por Γ um contexto de tipagem e por Σ um conjunto de definições indutivas, a relação de tipagem agora tem a forma $\Gamma;\Sigma \vdash t : T$. Por exemplo, assumindo que a definição indutiva de *nat* está em Σ, a seguinte relação de tipagem é válida: $\emptyset;\Sigma \vdash Z : nat$. Em particular, nós podemos usar o termo *nat_ind* para derivar o eliminador fraco *elim_nat* que usamos no começo da seção: $elim\_nat\,C\,f_Z\,f_S \equiv nat\_ind\ (\lambda n.C)\ f_Z\ (\lambda n.f_S)$. Podemos ver que de fato o eliminador fraco é uma especialização do caso geral denotado por *nat_ind*, em que a família $P : nat \to *$ é na verdade um só tipo para qualquer $n : nat$, constante, $C : *$. É isto que caracteriza *elim_nat* como um eliminador não-dependente.

Finalmente, o ponto (c) consiste simplesmente em generalizar a regra de conversão da seguinte maneira:

$$\frac{\Gamma;\Sigma \vdash t : \tau \qquad \Gamma;\Sigma \vdash \sigma : * \qquad \tau =_{\beta\iota} \sigma}{\Gamma \vdash t : \sigma} \text{ (5)}$$

ou seja, dois tipos $\beta\iota$-equivalentes têm os mesmos habitantes. Por exemplo, agora nós podemos provar a proposição $1 + 1 = 2$ no CIC com as definições indutivas de *nat* e *eq*. Basicamente, como o termo $1 + 1$ $\beta\iota$-reduz para 2, o cálculo reconhece que os termos $1 + 1 = 2$ e $2 = 2$ são $\beta\iota$-equivalentes, e portanto é trivial deduzir que $eq\_refl\,nat\,2$ habita o tipo $1 + 1 = 2$, e dessa forma verificar que a proposição $1 + 1 = 2$ é verdadeira.

Nós precisamos revisitar o conceito de ι-redução. No início da seção, nós apresentamos a ι-redução como uma regra de redução associada ao eliminador não-dependente do tipo *nat*. De fato, a ι-redução está associada diretamente ao princípio de indução do tipo indutivo. Por outro lado, nós também vimos que o eliminador não-dependente *elim_nat* é um termo derivado do princípio de indução *nat_ind*, e portanto a apresentação que vimos acima pode ser vista como uma simplificação do caso geral. A forma geral da ι-redução para o tipo *nat* é definida pelas

[4] Apesar da aparente complexidade do axioma *nat_ind*, em geral o princípio de indução pode ser derivado automaticamente a partir da declaração da família indutiva, e é por esse motivo que basta indicar o seu nome.

seguintes equações:

$$((nat\_ind\ P\ t\ f)\ Z) \rightarrow_\iota \quad t$$
$$((nat\_ind\ P\ t\ f)\ (S\ n)) \rightarrow_\iota \quad f\ n\ ((nat\_ind\ P\ t\ f)\ n)$$

O leitor pode conferir por conta própria que *elim_nat* ι-reduz como descrito no início da seção, quando definido a partir de *nat_ind* como descrito anteriormente, na presença das regras de redução acima.

4.3 A Natureza da Evolução do CIC

A motivação primordial para o desenvolvimento do CIC foi incorporar ao sistema formal a noção de tipo indutivos sem incorrer nos problemas técnicos apresentados no final da seção 3.4. A maneira encontrada para fazer isto foi introduzir conceitos como construtores e ι-redução ao cálculo, em oposição, por exemplo, à tentativa de explorar mecanismos de codificação[5]. Em outras palavras, a indução foi introduzida no cálculo de forma *ad hoc*. Esta particularidade explica porque este capítulo realiza toda uma discussão sobre o mecanismo de indução sem precisar generalizar conceitos vistos anteriormente.

Nesse ponto, é importante destacar que, apesar dos problemas mencionados na seção 3.4, nós vimos que é possível fazer aritmética no CC, bem como representar teorias de mais alta ordem. Além disso,é possível obter no CC o mesmo poder computacional que se tem no CIC (isto é, a recursão primitiva). E uma parte significativa desse poder computacional pode ser colocada a serviço da automatização do raciocínio, na regra de conversão. A maneira de fazer isto consiste em utilizar esquemas gerais de codificação, similares aos numerais de Church, para representar tipos indutivos arbitrários, que permitem automatização(PFENNING; PAULIN-MOHRING, 1989). Ou seja, se a diferença entre o CC e o CIC não está na capacidade expressiva do sistema, ou no seu poder computacional, então em que medida o CIC pode ser considerado uma evolução do CC?

Os pesquisadores que desenvolveram o CIC reconheceram a utilidade das famílias indutivas e, baseados em trabalhos anteriores sobre a indução, buscaram uma formulação direta deste conceito e integraram essa formulação ao CC. Na prática, isto significa que a noção de

[5] É importante deixar claro que a abordagem escolhida no CIC para trabalhar com indução não é necessariamente a melhor. Há pesquisa relacionada argumentando que a causa de certos defeitos em implementações do CIC e sistemas relacionados envolve tipos indutivos como primitivas dos sistemas, e defendendo o uso de novas técnicas de codificações mais avançadas que a usada para construir os numerais de Church, aliadas a extensões mais simples; veja (FIRSOV; STUMP, 2018)

tipos indutivos foi introduzida no sistema por construção. Isto é, sem a necessidade de buscar codificações que garantam todas as propriedades desejadas, pois o sistema já foi construído afim de capturar a indução adequadamente. Essa evolução pode ser contrastada com a extensão do CC ao STLC, onde o primeiro basicamente generaliza conceitos que já estavam presentes no segundo (e.g., o operador Π generaliza o conectivo $\rightarrow$). Do ponto vista teórico, essa estratégia é mais elegante que a introdução de conceitos por construção, pois ela herda boas propriedades do sistema. Nesse sentido, o que se pode dizer é que o CIC é uma evolução pragmática do CC.

5 COQ MODULO THEORY

A sequência de sistemas formais apresentada nos Capítulos 2,3,4 forma uma progressão, onde cada sistema possui mais capacidade computacional do que o anterior. Até agora, este ganho se deu pela associação de computação a elementos do sistema que antes só tinham caráter lógico: o CC associa computação às aplicações de função, e o CIC associa computação a termos indutivos. A extensão que nós vamos apresentar a seguir é de uma natureza um pouco diferente. O Coq Modulo Theory (CoqMT) expande a capacidade computacional do CIC estendendo a computação já existente nos tipos indutivos. Para fazer isso, o CoqMT identifica estruturas de primeira ordem nos termos do cálculo, e decide igualdades entre essas estruturas para auxiliar o processo de conversão.

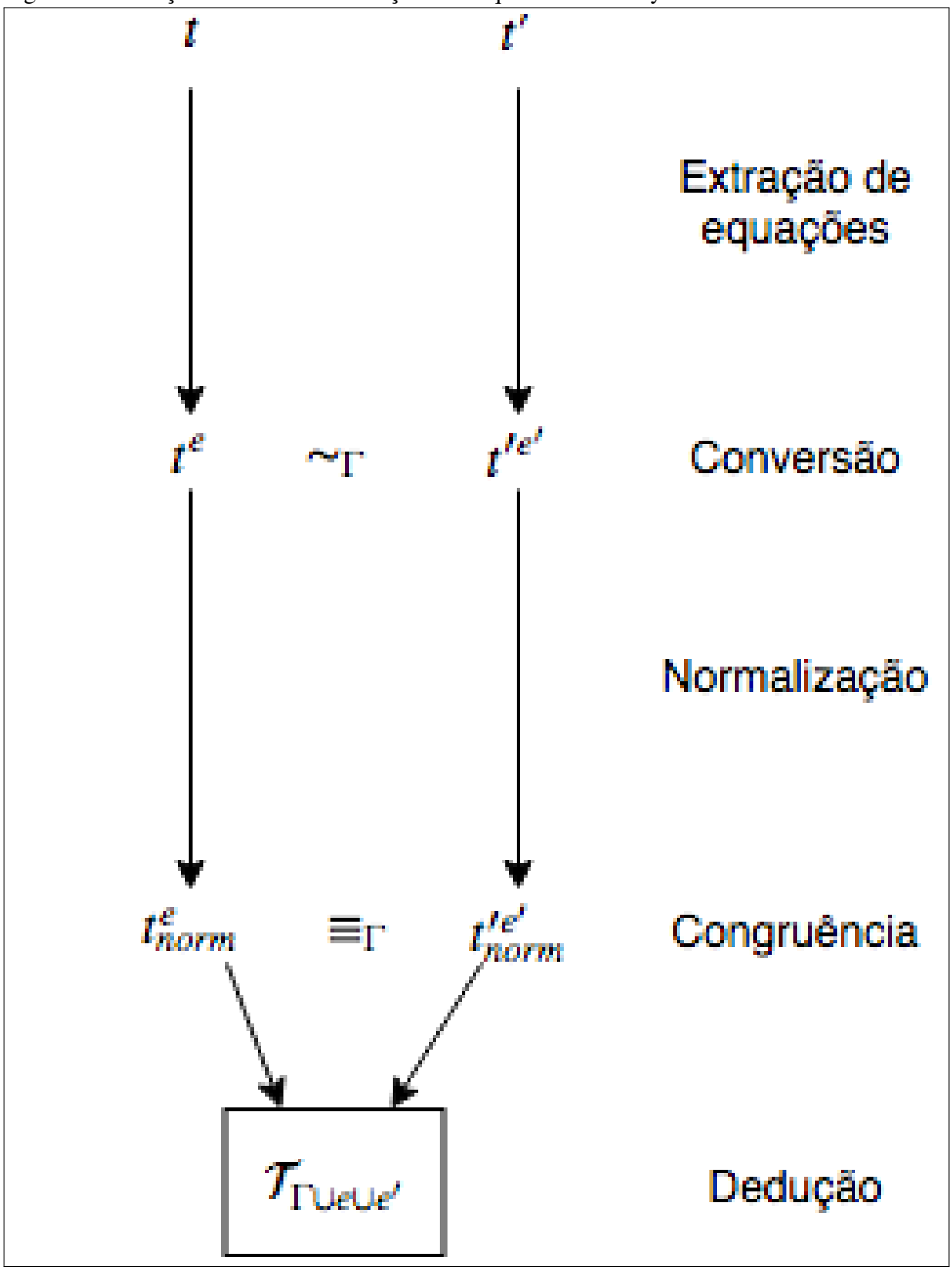
O cerne do CoqMT consiste na integração de uma (ou mais!) teoria de primeira ordem $\mathcal{T}$ (mais precisamente, de um procedimento de decisão para $\mathcal{T}$) ao processo de verificação de tipos do cálculo. Neste caso, $\mathcal{T}$ é usada como uma "caixa preta", de modo que algumas instâncias do problema de verificação de tipos são resolvidos com auxílio de $\mathcal{T}$. Dessa maneira, novos problemas passam a ser solucionáveis quando recorremos a $\mathcal{T}$. Mais do que isso, a caixa $\mathcal{T}$ integra raciocínio de primeira ordem ao cálculo, o que permite manipular termos que possuem estrutura de primeira ordem com mais flexibilidade.

5.1 Incorporando a teoria $\mathcal{T}$ ao sistema

A Figura 1 ilustra os principais aspectos técnicos dessa solução do ponto de vista operacional:

A parte de cima da figura ilustra o fato de que no momento da escrita dos termos, certas equações são extraídas para auxiliar o processo de dedução em $\mathcal{T}$. Durante o processo de verificação de tipos, o sistema eventualmente chega a um problema de conversão: $T \sim_E T'$. Nesse ponto os termos T e T' são normalizados, o que pode envolver uma série de reduções e invocações de $\mathcal{T}$, o que dá origem ao seguinte problema de congruência: $T^* \equiv_E T'^*$. Esse problema de congruência, por sua vez, pode eventualmente ser resolvido pela teoria $\mathcal{T}$ com auxílio do conjunto E de equações extraídas. As subseções a seguir vão detalhar cada passo desse processo.

Figura 1 – Esboço do fluxo de informação no Coq Modulo Theory



Source: o autor.

5.1.1 Algebrização

O processo de extração das equações que irão auxiliar a dedução em $\mathcal{T}$ requer a conversão de termos do cálculo em termos de $\mathcal{T}$: isto é chamado de algebrização. Por exemplo, o resultado da algebrização do termo $\mathbf{S}x+\mathbf{0}$ do cálculo é o $\mathcal{T}$-termo $S(x)+0$ (note a mudança de fonte para indicar a mudança de linguagem). Um exemplo mais interessante é dado por $\mathbf{S}t+\mathbf{0}$, onde t é um termo arbitrário complexo do cálculo. Nesse caso, o resultado da algebrização é o $\mathcal{T}$-termo $S(y)+0$, onde y é uma variável de primeira ordem nova que tem o papel de abstrair o termo t. Ou seja, o processo de algebrização simplesmente extrai a estrutura algébrica de um termo do cálculo. As definições abaixo formalizam esta ideia.

Definição 5.1 (Contexto Algébrico)**.** *Um contexto algébrico é essencialmente um termo da aritmética de Presburger. Isto é, um termo gerado pela seguinte gramática:*

$$C_{\mathcal{A}} ::= \mathbf{0} \mid \mathbf{S}\ (C_{\mathcal{A}}) \mid C_{\mathcal{A}}+C_{\mathcal{A}} \mid x$$

onde x é uma variável de primeira ordem, e não se permite o casoem que $C_{\mathcal{A}}$ se reduz a uma única variável.

Considere um contexto algébrico $C_{\mathcal{A}}$ que envolve apenas as variáveis $x_1, x_2, \ldots, x_n$. Escrevemos $C_{\mathcal{A}}[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ para indicar o termo obtido pela substituição de cada variável x_i por t_i. O construto $[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ é chamado de *instanciação de contexto*, e será comumente denotado pelas letras $\mathcal{I}, \mathcal{J}$, chamadas *variáveis de instanciação*.

Definição 5.2 (Termo pré-algébrico)**.** *Um termo t do cálculo é* pré-algébrico *se ele é da forma $C_{\mathcal{A}}[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$. Se nenhum dos termos t_i é pré-algébrico, nós dizemos que $C_{\mathcal{A}}$ é a* descrição algébrica maximal *de t.*

Por exemplo, o termo $(\mathbf{S}t)+u$ é pré-algébrico. Nós podemos tomar x_1+x_2 como contexto algébrico juntamente com a instanciação $[x_1 \leftarrow \mathbf{S}t, x_2 \leftarrow u]$. Este contexto, no entanto, não é maximal. Assumindo t e u não encabeçados por um símbolo de $\mathcal{T}$ (e portanto não sendo pré-algébrico), $S\ (x_1)+x_2$ é a descrição algébrica maximal de $(\mathbf{S}t)+u$, juntamente com a instanciação $[x_1 \leftarrow t, x_2 \leftarrow u]$.

Definição 5.3 (Algebrização)**.** *Uma* algebrização *de um termo t é um par $(C, \mathcal{I})$ tal que $t = C[\mathcal{I}]$. Se C é a descrição algébrica maximal de t, nós dizemos que $(C, \mathcal{I})$ é a* algebrização maximal *de t.*

5.1.2 Normalização

A primeira etapa da solução de um problema de conversão $T \sim_E T'$ consiste em normalizar os termos T e T' através de uma série de reduções e invocações de $\mathcal{T}$. O CoqMT utiliza a teoria $\mathcal{T}$ no processo de redução dos termos, introduzindo a noção de $\beta\iota\mathcal{T}$-redução. A ideia aqui é que a teoria $\mathcal{T}$ pode oferecer uma função de normalização $\text{norm}_{\mathcal{T}}$. Por exemplo, Shostak (1979) apresenta um método que pode ser usado para definir uma função de normalização para os termos da aritmética de Presburger. Essa função de normalização é encapsulada no cálculo através da seguinte definição:

Definição 5.4 ($\mathcal{T}$-normalização)**.** *Seja $t = C[\mathcal{I}]$ um termo CoqMT com contexto algébrico maximal C, e $\mathcal{I} = [y_1 \leftarrow t_1, \ldots, y_n \leftarrow t_n]$. Seja $\mathcal{J} = [y_1 \leftarrow x_1, \ldots, y_n \leftarrow x_n]$ onde todos os x_i's são diferentes aos pares, e não sejam livres nos t_i's. A $\mathcal{T}$-normalização de t, escrita* $\mathbf{norm}(t)$*, é definida como:*

$$\mathbf{norm}(t) = \text{norm}_{\mathcal{T}}(t[\mathcal{J}])\{x_1 \to t_1\}\cdots\{x_n \to t_n\}$$

A função **norm** é utilizada nas regras de ι-redução da seguinte maneira:

(ι_0) $\text{Elim}(t : Q)\{f_0, f_S\} \to f_0$ se t é pré-algébrico, e $\mathbf{norm}(t) = 0$

(ι_S) $\text{Elim}(t : Q)\{f_0, f_S\} \to f_S\, v\ \text{Elim}(v : Q)\{f_0, f_S\}$ se t é pré-algébrico, e $\mathbf{norm}(t) = \mathbf{S}\, v$

Essas regras são essencialmente as mesmas que vimos no capítulo 4, mas agora elas permitem aplicar a redução em situações em que antes não seria possível. Por exemplo, assumindo que $\text{norm}_{\mathcal{T}}((x + S(0)) + y) = S(x + y)$, nós temos que $\mathbf{norm}((u + (\mathbf{S}\,\mathbf{0})) + u') = \mathbf{S}\ (u + u')$. Isso implica que o termo $\text{Elim}((x + (\mathbf{S}\,\mathbf{0})) + y : Q)\{f_0, f_S\}$, que é normal para a ι-redução padrão, agora reduz para $f_S\ (x + y)\ \text{Elim}((x + y) : Q)\{f_0, f_S\}$.

5.1.3 Equações extraídas

Como mencionamos no início, o processo de dedução em $\mathcal{T}$ é auxiliado por um conjunto de equações que foram extraídas durante a construção dos termos. A seguir, nós vamos ver como esse auxílio é implementado.

O primeiro passo é entender de onde vêm as equações extraídas. Quando o usuário escreve o termo função $\lambda\,[p : x = 0].\gamma$, onde γ é um termo qualquer, ele pode vir a fazer as seguintes observações: (1) O parâmetro p tem tipo igualdade, (2) A verificação de tipo do corpo γ da função depende do fato de que o tipo igualdade $x = 0$ é habitado (a saber, pelo parâmetro

p), (3) Mais especificamente, esse fato é necessário para alguma verificação de congruência (auxiliada por $\mathcal{T}$), mas aqui ele não está disponível. Nesse momento, o usuário pode extrair a equação $x = 0$, reescrevendo o termo como $\lambda\left[p :^{x=0} x = 0\right].\gamma$, de modo que $\mathcal{T}$ passa a ter acesso a essa informação. Da mesma forma que os termos função, termos formados com o quantificador Π também oferecem uma oportunidade para a extração de equações. Por exemplo, $\Pi\left[q :^{x=1} x = 1\right].\eta$.

A seguir, lembre que $\mathcal{T}$ é um procedimento de decisão para uma teoria de primeira ordem. Portanto, as equações a serem extraídas devem ter o formato de equações de primeira ordem, que são obtidas a partir das equações do cálculo pelo processo de algebrização. Assim, formalmente, as equações extraídas são definidas da seguinte maneira:

Definição 5.5 (Equação extraída)**.** *Uma* equação extraída *é uma quádrupla* $(C_1, C_2, \mathcal{I}_1, \mathcal{I}_2)$ *(escrita* $(C_1 = C_2, \mathcal{I}_1, \mathcal{I}_2)$ *tal que* C_1 *e* C_2 *são contextos algébricos e* $\mathcal{I}_1, \mathcal{I}_2$ *são instanciações de contexto.*

Agora, nós estamos prontos para ver como as equações extraídas são utilizadas por $\mathcal{T}$. Considere novamente o termo $\lambda\left[p :^{x=0} x = 0\right].\gamma$[1]. Durante o processo de verificação de tipo desse termo função, considera-se a tipagem do termo γ em um contexto Σ que contém a relação de tipagem anotada $\left[p :^{x=0} x = 0\right]$. Dessa maneira, invocações de $\mathcal{T}$ para decidir congruências fornecem a equação $x = 0$, para que $\mathcal{T}$ possa raciocinar utilizando essa informação. De fato, todas as equações extraídas que se encontram no contexto Σ nesse momento são passadas para $\mathcal{T}$.

5.1.4 Congruência

Finalmente, nós chegamos ao último passo do esquema: a relação de congruência $\equiv^b_E$. Como vimos acima, no Cálculo das Construções Indutivas (Coq) a relação de congruência é estabelecida pela mera verificação sintática da identidade entre dois termos. No entanto, para integrar a teoria $\mathcal{T}$ ao restante do sistema, o CoqMT introduz um cálculo formal para a relação de congruência. Existe uma primeira regra que estabelece a congruência sintática entre dois termos:

$$\frac{}{t \equiv^b_E t}\ (\text{Refl})$$

[1] Estritamente falando, de acordo com a definição 5.5, esse termo deveria ser escrito como $\lambda\left[p :^{(q,0,[q\to x],[])} x = 0\right].\gamma$, mas nós vamos continuar com a notação simplificada para facilitar a leitura.

E existem regras para estabelecer as congruências derivadas da teoria $\mathcal{T}$:

$$\frac{t = u \text{ é uma } (\mathcal{T}, \emptyset)\text{-consequência módulo a relação vazia}}{t \equiv_E^{\perp} u} \text{ (Ded}\perp\text{)}$$

$$\frac{t = u \text{ é uma } (\mathcal{T}, E)\text{-consequência módulo } \overset{*}{\leftrightarrow} \cdot \equiv_E^b \cdot \overset{*}{\leftrightarrow}}{t \equiv_E^{\top} u} \text{ (Ded}\top\text{)}$$

Finalmente, existem regras que permitem propagar congruências elementares para congruências entre termos complexos. Por exemplo:

$$\frac{U \equiv_E^b U' \qquad v \equiv_{\{e\} \cup E}^b v' \qquad e \equiv_E^b e'}{\lambda\,(x :^e U).v \equiv_E^b \lambda\left(x :^{e'} U'\right).v'} \text{ (Lam-}\varepsilon\text{)}$$

Em particular, a regra Lam-ε também ilustra como as equações anotadas e são passadas para o conjunto de equações extraídas E durante a derivação. Na próxima seção, nós veremos em detalhe a propagação e processamento de equações extraídas em diversos pontos do sistema.

Note que o cálculo possui duas regras para derivar congruências entre termos utilizando a teoria $\mathcal{T}$. A diferença básica entre elas é que uma utiliza as equações extraídas do contexto (Ded $\top$) enquanto a outra é limitada a congruências deriváveis a partir de contexto vazio (Ded $\perp$). A razão para essa separação é que existem contextos (e.g. raciocínio por contradição) em que as congruências derivadas a partir do conjunto de equações extraídas E podem ter impacto negativo sobre as boas propriedades meta-teóricas do sistema. Na prática, o cálculo usa a regra (Ded $\top$) sempre que possível, utilizando a outra regra em contextos onde é preciso realizar uma derivação mais segura. Para mais detalhes, veja (STRUB, 2010, p.9).

Abaixo nós temos as definições que formalizam a noção de $(\mathcal{T}, E)$-consequência.

Definição 5.6. *Seja $\{C_i\}_{1 \leq i \leq n}$ um conjunto de contextos pré-algébricos, $\{\mathcal{I}_i\}_{1 \leq i \leq n}$ um conjunto de instanciações de contexto e $\mathcal{R}$ uma relação binária entre termos CoqMT. Para qualquer $i \in \{1..n\}$, k, seja $t_{i,k}$ o termo associado à k-ésima variável pela instanciação $\mathcal{I}_i$.*

Seja $\{\mathcal{J}_i\}_{1 \leq i \leq n}$ um conjunto de instanciações e $\mathcal{V}$ um conjunto de equações entre variáveis. Dizemos que $\{\mathcal{J}_i\}_i$ abstrai $\{(C_i, \mathcal{I}_i)\}$ de acordo com $\mathcal{V}$ e $\mathcal{R}$ se:

1. *Para todo $i \in \{1..n\}, k$, $\mathcal{J}_i$ associa uma variável nova $x_{i,k}$ à k-ésima variável de C_i*
2. *$\left(x_{i,p} = x_{j,q}\right) \in \mathcal{V}$ implica em $t_{i,p} \mathcal{R} t_{j,q}$*

Definição 5.7 ($(\mathcal{T}, E)$-consequência)**.** *Seja $E = \left[\left(C_i^l = C_i^r, \mathcal{I}_i^l, \mathcal{I}_i^r\right) \mid 1 \leq i \leq n\right]$ uma sequência de equações extraídas, e t, u dois termos CoqMT da forma $C_0^l\left[\mathcal{I}_0^l\right]$ e $C_0^r\left[\mathcal{I}_0^r\right]$ respectivamente. Seja $\mathcal{R}$ qualquer relação binária em termos CoqMT.*

Nós dizemos que $(t = u)$ *é uma* $(\mathcal{T}, E)$*-consequência modulo* $\mathcal{R}$ *se existe um conjunto* $\left\{\mathcal{J}_i^\alpha\right\}_{i,\alpha}$ *de instanciações de contexto e um conjunto* $\mathcal{V}$ *de equações entre variáveis tais que*

1. $\mathcal{T}, \left\{C_i^l\left[\mathcal{I}_i^l\right] = C_i^r\left[\mathcal{I}_i^r\right]\right\}, \mathcal{V} \vDash C_0^l\left[\mathcal{I}_0^l\right] = C_0^r\left[\mathcal{I}_0^r\right]$,
2. $\left\{\mathcal{J}_i^\alpha\right\}_{i,\alpha}$ *abstrai* $\left\{(C_i^\alpha, \mathcal{I}_i^\alpha)\right\}_{i,\alpha}$ *de acordo com* $\mathcal{V}$ *e* $\mathcal{R}$.

5.2 Aspectos Técnicos

Na discussão acima, nós vimos que as equações extraídas aumentam as oportunidades de dedução que podem ser realizadas pela teoria $\mathcal{T}$. No entanto, o usuário precisa indicar que equações ele deseja extrair (mais adiante, na seção 5.3, nós vamos discutir um caso em que a extração de uma equação acarreta em problemas). Como vimos também, as equações a serem extraídas são opcionalmente anotadas no momento que o usuário escreve termos λ ou Π. A questão é que as equações só são utilizadas no passo de congruência, e existe um longo caminho para chegar até lá. Mais precisamente, as anotações são feitas em termos que são decompostos e/ou transformados por aplicações de regras de tipagem, computação e congruência, e é preciso tomar cuidado para que a informação contida na anotação não se perca. A seguir nós vamos discutir vários pontos onde essas anotações podem ser perdidas, e apresentar os recursos adicionais que foram introduzidas para evitar que isso aconteça.

1. Considere o termo α definido como $\left(\lambda\left[p :^{x=0} x = 0\right].\gamma\right)\ t$, ou seja, a aplicação de um termo função, com uma variável anotada, a um termo arbitrário t. Suponha que durante uma verificação qualquer de tipos $\Sigma \vdash h : H$, onde H contém o termo α, é aplicada uma β-redução ao termo α. Como sabemos, o termo que resulta dessa redução é $\gamma\{p \leftarrow t\}$. O problema aqui é que a anotação da equação extraída foi perdida. Para resolver essa dificuldade, introduz-se outra forma de anotação para equações extraídas, o *marcador de equação* $\bullet\left[u :^{\varepsilon} U\right]$, onde u e U são termos arbitrários e ε é a equação extraída a partir de U. Agora basta alterar a regra de β-redução para produzir o termo $\bullet\left[t :^{x=0} x = 0\right].\gamma\{p \leftarrow t\}$. Abaixo, temos a definição formal dessa versão da β-redução:

$$(\beta)\left(\bullet\left[\overrightarrow{w :^{e} W}\right].\lambda\left[x : U\right].v\right)\ u \to \bullet\left[\overrightarrow{w :^{e} W}\right].v\{x \leftarrow u\}$$
$$(\beta - \mathcal{E})\left(\bullet\left[\overrightarrow{w :^{e} W}\right].\lambda\left[x :^{e'} U\right].v\right)\ u \to \bullet\left[\overrightarrow{w :^{e} W}\right].\bullet\left[u :^{e'} U\right].v\{x \leftarrow u\}$$

   Ou seja, a primeira regra simplesmente preserva o prefixo de anotações $\bullet\left[\overrightarrow{w :^{e} W}\right]$ de um termo função sem anotação na variável, e a segunda regra acrescenta a anotação do termo função a esse prefixo.

2. Além da β-redução, o cálculo de tipagem também precisa ser adequadamente reformulado afim de preservar anotações ao longo da derivação do tipo de um termo anotado. O primeiro passo consiste em mostrar como surgem contextos com atribuições de tipagem anotadas, como aludido na seção 5.1.3. As regras que introduzem termos com equações extraídas anotadas à árvore de derivação tem o papel de verificar a correspondência entre uma equação extraída e o tipo associado. Estas regras tem como hipótese a extraibilidade da equação, definida a seguir:

   Definição 5.8 (E-Extraibilidade)**.** *Seja $e \equiv (C_1 = C_2, \mathcal{I}, \mathcal{J})$ uma equação extraída, T um termo, e E um conjunto de equações extraídas. Nós dizemos que e é E-*extraível *a partir de T quando $T \sim_E C_1[\mathcal{I}] = C_2[\mathcal{J}]$.*

   Veja abaixo um exemplo de regra de introdução:

$$\frac{x \in V \qquad \Gamma \vdash \tau : s \qquad e \text{ é } \Gamma\text{-extraível a partir de } \tau}{\Gamma \cup \{x :^e \tau\} \vdash x : \tau} \text{ (Var)}$$

   Note que quando não há uma equação extraída, nós simplesmente desconsideramos a última hipótese, obtendo a mesma regra que usamos desde o Cálculo das Construções.
3. Num contexto $\Sigma \equiv \Gamma \cup \{p :^{x=0} x = 0\}$, suponha ser possível derivar $\Sigma \vdash \gamma : (G\,x)$, onde $G : \mathbb{N} \to *$ é uma família qualquer. Nesse contexto, também é possível derivar $\Sigma \vdash \gamma : (G\,0)$, pois $(G\,x) \sim_\Sigma (G\,0)$ (i.e., $(G\,x)$ e $(G\,0)$ são conversíveis no contexo Σ que contém $x = 0$) possibilita a aplicação da regra de conversão para realizar tal derivação. A partir disto, nós podemos formar o termo função $(\lambda\,[p :^{x=0} x = 0]\,.\gamma)$. Ignorando a anotação, este termo teria tipo da forma $\Pi p : (x = 0)\,.\,(G\,x)$ ou $\Pi p : (x = 0)\,.\,(G\,0)$ (isto é, seria possível derivar ambos os tipos para a função). A dificuldade aqui encontrada é que a aplicação da regra de conversão entre estes dois tipos não é mais possível. Lembre que a derivação de um tipo função ocorre num contexto menor (neste caso, Γ), sem a atribuição de tipagem correspondente à variável da função. Como o contexto Γ não contém a equação $x = 0$, e a regra de conversão tem acesso apenas ao contexto e aos tipos envolvidos, a conversão entre os tipos função apresentados acima não é possível. A solução para este problema consiste em reproduzir a anotação do termo função também no seu tipo, da seguinte maneira:

$$(\lambda\,[p :^{x=0} x = 0]\,.\gamma) : \Pi p :^{x=0} (x = 0)\,.G$$

   Agora, as regras que eliminam atribuições de tipagem devem preservar as anotações nos

tipos, e ficam da seguinte maneira:

$$\frac{\Gamma \cup \left\{x :^{w?} \tau\right\} \vdash e : \sigma \qquad \Gamma \vdash \left(\Pi x :^{w?} \tau.\sigma\right) : s}{\Gamma \vdash \left(\lambda x :^{w?} \tau.e\right) : \left(\Pi x :^{w?} \tau.\sigma\right)} \text{ Lam}$$

$$\frac{\Gamma \vdash \sigma : s_1 \qquad \Gamma \cup \left\{x :^{w?} \sigma\right\} \vdash \tau : s_2}{\Gamma \vdash \left(\Pi x :^{w?} \sigma.\tau\right) : s_2} \text{ Prod}$$

A única diferença destas regras para as regras análogas no CC e no CIC é a presença da anotação opcional $w?$ (que, quando presente, deve estar presente em todos os termos da regra).

4. Considere novamente a aplicação do termo $\left(\lambda \left[p :^{x=0} x = 0\right].\gamma\right)$ ao termo t, que ilustrou a discussão um pouco mais acima sobre β-redução. No que diz respeito ao tipo destes termos, t deve ter tipo $x = 0$, para que seja possível formar a aplicação. Já o tipo da função depende do tipo de γ, que não conhecemos. Vamos dizer que $\gamma : G$. O tipo da função $\left(\lambda \left[p :^{x=0} x = 0\right].\gamma\right)$ é, neste caso, $\Pi p :^{x=0} (x = 0).G$. Mas, quando desejamos encontrar o tipo da aplicação α, nós observamos um problema análogo ao que ocorre na β-redução, em que uma anotação numa vinculação se perde; se considerarmos apenas $G\{p \leftarrow t\}$ como o tipo de $\left(\lambda \left[p :^{x=0} x = 0\right].\gamma\right) t$, a anotação $x = 0$ não está presente. Este problema é solucionado da mesma forma. A anotação é reintroduzida através de uma alteração da regra de aplicação para incluí-la no tipo final. Abaixo, temos a definição formal da regra[2]:

$$\frac{\Gamma \vdash t_1 : \bullet \left[\overrightarrow{w :^{e} W}\right].\Pi x :^{g} \sigma.\tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash (t_1\, t_2) : \bullet \left[\overrightarrow{w :^{e} W}\right].\bullet \left[t_2 :^{g} \sigma\right].(\tau\,[x := t_2])} \text{ App-}\varepsilon$$

Note a semelhança entre esta regra de aplicação, e o que foi definido como β-redução para o CoqMT.

5. A seguir, nós vamos apresentar duas regras adicionais que foram introduzidas no cálculo apenas para resolver problemas relacionados com termos $\bullet$. Considere um contexto Γ, onde $\Gamma \cup \left\{p :^{x=0} x = 0\right\} \vdash \gamma : G$, no qual as derivações de tipo a seguir são realizadas. A primeira derivação consiste numa aplicação da regra App-ε ao termo função $\left(\lambda \left[p :^{x=0} x = 0\right].\gamma\right)$ de tipo $\Pi \left(p :^{x=0} x = 0\right).G$ e argumento t de tipo $x = 0$.

$$\frac{\Gamma \vdash \left(\lambda \left[p :^{x=0} x = 0\right].\gamma\right) : \Pi \left(p :^{x=0} x = 0\right).G \qquad \Gamma \vdash t : x = 0}{\Gamma \vdash \left(\left(\lambda \left[p :^{x=0} x = 0\right].\gamma\right) t\right) : \bullet [t : x = 0].(G[p := t])} \text{ App-}\varepsilon$$

[2] Assim como a regra de β-redução foi dividida em duas, a regra de aplicação também foi dividida em duas, onde a variante que não apresentamos simplesmente preserva o prefixo de anotações.

Por outro lado, segue que o mesmo termo $\left(\left(\lambda\left[p :^{x=0} x = 0\right].\gamma\right)\ t\right)$ pode ser β-reduzido para $\bullet\left[t : x = 0\right].\left(\gamma[p := t]\right)$. Nós devemos esperar então que o termo β-reduzido também tenha tipo $\bullet\left[t : x = 0\right].\left(G[p := t]\right)$. Ou seja, nós gostaríamos de fazer uma segunda derivação, da forma:

$$\Gamma \vdash \bullet\left[t : x = 0\right].\left(\gamma[p := t]\right) : \bullet\left[t : x = 0\right].\left(G[p := t]\right)$$

A regra que é apresentada no CoqMT para realizar esta derivação é a seguinte:

$$\frac{\Gamma \cup \{\bullet\left[t :^{e} \tau\right]\} \vdash u : U \qquad \Gamma \vdash t : \tau \qquad e \text{ é } \Gamma\text{-extraível a partir de } \tau}{\Gamma \vdash \bullet\left[t :^{e} \tau\right].u : \bullet\left[t :^{e} \tau\right].U}\ \bullet$$

Na aplicação desta regra ao nosso exemplo, a hipótese chave é derivar $\gamma[p := t] : G[p := t]$, mas no contexto contendo $\bullet\left[t : x = 0\right]$. É importante observarmos porquê. Quando derivamos γ originalmente (como o corpo da função $\left(\lambda\left[p :^{x=0} x = 0\right].\gamma\right)$), o contexto continha a equação $x = 0$ através da atribuição de tipagem $\left\{p :^{x=0} x = 0\right\}$. Mas na derivação de $\gamma[p := t]$, nós sabemos apenas que é possível construir $t : x = 0$, e não que a equação pode ser extraída no contexto. A solução apresentada no CoqMT é dada pela seguinte regra:

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash u : U \qquad e \text{ é } \Gamma\text{-extraível a partir de } \tau}{\Gamma \cup \{\bullet\left[t :^{e} \tau\right]\} \vdash u : U}\ \text{Weak-}\bullet$$

Ou seja, se num contexto Γ é possível derivar $u : U$, então também é possível realizar a mesma derivação no contexto estendido com a equação e, assumindo que tal equação é de fato extraível em Γ. Esta é a única regra que introduz termos $\bullet$ ao contexto. Isto significa que não há uma regra primitiva que permita a construção de um termo que dependa diretamente da presença de marcadores de equação no contexto (ao contrário de atribuições e a regra Var), e o uso dessas equações no contexto é apenas aquele indicado pelo exemplo, isto é, tornar visível para a regra de conversão que a equação e é extraível no contexto. Além disso, esta é a última regra do cálculo que lida diretamente com marcadores de equação. De fato, nós apresentamos todas as regras do CoqMT(STRUB, 2010), e estamos preparados para realizar uma discussão sobre as implicações da presença dos termos $\bullet$, um pouco mais adiante, na seção 5.3.

Finalmente, além das modificações e regras adicionais descritas acima, o CoqMT também redefine a noção de contexto de tipagem. Como vimos acima, o contexto também contém termos $\bullet$. O contexto de tipagem é, portanto, uma coleção de atribuições de tipagem opcionalmente anotadas $x :^{e?} T$, juntamente com marcadores de equação $\bullet\left[t :^{e} T\right]$. Agora que temos a definição

de contexto, nós podemos apresentar a definição formal da regra de conversão:

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash \sigma : * \qquad \tau \sim_\Gamma \sigma}{\Gamma \vdash t : \sigma}\ \text{Conv}$$

onde $\sim_\Gamma$ é definido como $\overset{*}{\leftrightarrow} \cdot \equiv^b_E \cdot \overset{*}{\leftrightarrow}$, onde E é o conjunto de todas as equações extraídas presentes em Γ . Por sua vez, o processo de congruência também pode adicionar novas equações ao conjunto E. Nós vimos na seção 5.1.4 como isto é feito por meio da regra Lam-ε, mas as equações também podem ser adicionadas de maneira semelhante por meio de outras regras.

5.3 Conclusão

Como mencionamos na introdução deste capítulo, a discussão que apresentamos foi baseada no artigo *Coq Modulo Theory*(STRUB, 2010). No contexto de uma extensiva pesquisa de seu autor sobre como incorporar teorias decidíveis de primeira ordem ao Cálculo de Construções Indutivas, este trabalho representa um ponto intermediário, no qual a ideia de equações extraídas aparece na sua forma mais bem-acabada. Mais especificamente, as versões anteriores do trabalho(STRUB, 2008; BLANQUI *et al.*, 2007) ainda não continham o recurso dos marcadores de equação $\bullet$, e impediam a redução dos termos quando isso acarretava na perda de uma equação extraída, como relatado em (STRUB, 2010). Por outro lado, os trabalhos mais recentes(BARRAS *et al.*, 2011; JOUANNAUD; STRUB, 2017) se concentram na demonstração de propriedades meta-teóricas, em uma versão do sistema que não trabalha com equações extraídas. Portanto, o artigo que estudamos corresponde à versão do trabalho que busca fazer o uso mais extensivo da teoria $\mathcal{T}$, no sentido de aumentar a automação oferecida pelo sistema, que é o ponto central de nosso interesse.

Na realidade, o abandono das equações extraídas que se observa nos trabalhos mais recentes é motivado também por problemas técnicos relacionados com a ocorrência dos marcadores de equação $\bullet$ nos tipos. O exemplo a seguir ilustra um problema que pode ocorrer. Suponha que num contexto Γ seja possível derivar $\Gamma \vdash (\lambda x : T.u) : (\Pi x : T.U)$, onde não há ocorrências de x em U. Considere também que é possível derivar $\Gamma \vdash f : U \to B$, e $\Gamma \vdash t : T$. Então, claramente é possível derivar um termo com tipo B, sendo este simplesmente $(f\ ((\lambda x : T.u)\ t))$. A seguir, suponha que o tipo T tenha sido extraído durante a derivação de u, resultando em $\Gamma \cup \{x :^e T\} \vdash u : U$. Portanto a função formada é $\Gamma \vdash (\lambda x :^e T.u) : (\Pi x :^e T.U)$. Mas isto significa que, quando formamos a aplicação $((\lambda x :^e T.u)\ t)$, esta terá tipo $\bullet\,[t :^e T]\,.U$, e não pode ser aplicada à função f pois esta função recebe argumentos do tipo U. Este exemplo ilustra o fato

de que, se por um lado a extração de equações aumenta as oportunidades de dedução em $\mathcal{T}$, por outro lado, elas podem deixar marcações $\bullet$ nos tipos dos termos resultantes, que podem gerar incompatibilidades. Isto mostra que a extração de equações deve ser feita com cuidado, o que inviabiliza a construção de mecanismos ingênuos de extração automática. O autor aponta também para a possibilidade de adotar o recurso de sub-tipagem para remover as marcações de equações nos tipos(STRUB, 2010).

Na prática, a solução adotada pelo autor foi remover completamente o recurso de equações extraídas do CoqMT. Nos artigos que vieram em seguida, não são feitas maiores considerações sobre o assunto, e nesse sentido, não é claro se a decisão foi motivada pelos problemas mencionados acima, ou se a ideia era facilitar a demonstração de propriedades meta-teóricas, mas o fato é que as equações extraídas foram efetivamente removidas. A questão que se coloca é, portanto, o que resta do CoqMT sem equações extraídas. Basicamente, nós podemos entender o sistema em termos de duas características relativamente independentes: A) uma função de normalização, utilizada em conjunto com a ι-redução, e B) uma versão limitada de dedução na teoria $\mathcal{T}$, limitada a equações válidas sem o uso de hipóteses em $\mathcal{T}$. Esta divisão se vê claramente no que foi apresentado[3], onde há separadamente uma função norm e a regra Ded$\bot$ (note que a regra Ded$\top$ não é necessária na ausência de equações extraídas).

Sem a possibilidade de utilizar equações extraídas, todas as igualdades demonstradas por uso direto da teoria $\mathcal{T}$ são igualdades livre de contexto. Há uma conexão imediata com as ideias presentes nas teorias de tipo onde há a presença da regra da extensionalidade; nós podemos ver que o CoqMT torna a conversão do sistema mais poderosa também através da adição de igualdades simples. Mas nestas teorias, a indecidibilidade mostra-se como um problema sério, visto que não há nenhuma orientação sobre como exatamente tais igualdades devem ser utilizadas. Nós vemos que as reduções que estudamos em sistemas anteriores não sofrem desse problema, pois carregam consigo uma *direção*, observada na simplificação estrutural dos termos reduzidos. Quando consideramos igualdades arbitrárias, ou mesmo aquelas expressíveis em primeira ordem, não é necessariamente o caso que tal direção exista (considere por exemplo a comutatividade $n+m=m+n$). O CoqMT contorna este problema exigindo a decidibilidade como pré-condição, ficando a cargo do algoritmo de decisão como devem ser aplicadas as igualdades básicas da teoria. Mas o outro aspecto remanescente do CoqMT oferece uma solução mais elegante. A

[3] Isto contrasta com versões mais recentes, porque nestas foi feita uma reformulação afim de definir a função de normalização em termos da relação de congruência. Isto significa que há uma simplificação em tais versões, onde ao invés da função definida independentemente, há apenas o caso restrito em que a função é, num certo sentido, “trivial”.

função de normalização é definida afim de expor um construtor para que a ι-redução possa operar sobre o termo normalizado. Se encaixarmos o uso de igualdades numa *estrutura indutiva* nós podemos fazer uso desta para direcionar tais igualdades da mesma forma que as reduções são direcionadas. Desta forma, tratamos de uma coleção de igualdades num só movimento, diferindo de teorias extensionais (em que cada igualdade é considerada isoladamente), mas sem a obrigação de lidar com uma teoria inteira como no CoqMT. O que exatamente se entende por estrutura indutiva, assim como o encaixe em questão, são tópicos de discussão do próximo capítulo.

6 CONCLUSÃO

Lembre da pergunta que foi deixada na Introdução: como transformar conteúdo matemático em regras de computação, de modo que a automação no sistema possa crescer de maneira gradual e construtiva, a medida que os teoremas vão sendo provados. Para começar a responder essa pergunta, nós vamos reexaminar alguns aspectos associados com a computação presente nos sistemas formais que vimos nos capítulos anteriores. O Cálculo das Construções (CC) é o primeiro ponto da sequência estudada em que de fato passa a ocorrer alguma forma de automação. Basicamente, introduz-se a noção de conversão, que consiste em verificar se dois tipos são equivalentes por meio de computação. Em princípio, essa computação se resume a aplicação de β-redução, mas é possível incorporar outras computações por meio de codificação. Essa ideia de um mecanismo de conversão automático nos parece bastante conveniente. Além disso, nós vimos que no CC é possível codificar tipos indutivos e a computação associada a eles (e.g. números naturais, através da codificação de Church). No entanto, a abordagem baseada em codificação tem algumas limitações sérias, como ineficiência e falta de expressividade (veja as seções 4.3 e 3.4, respectivamente). Esses problemas parecem indicar que a codificação não é um bom caminho para obter computação mais expressiva. De fato, o CIC oferece uma outra solução para implementar computação indutiva, ao incorporar ao sistema uma noção específica de tipos indutivos, e associar computação via ι-redução a eles. Além de resolver as limitações indicadas acima, isso realmente oferece uma solução bem mais elegante. Finalmente, o CoqMT tenta ir além, ao propor a possibilidade de incorporar uma teoria $\mathcal{T}$ de primeira ordem diretamente ao processo de conversão. Em princípio, isto parece uma boa ideia, mas a implementação que estudamos apresenta uma série de problemas de ordem prática, por ter caráter muito geral. Por outro lado, ao restringir as possibilidades de aplicação da teoria, nos parece que a função de normalização de $\mathcal{T}$ acaba se tornando a parte mais importante do sistema. Como no CoqMT a função de normalização está acoplada à regra de ι-redução, nós temos mais uma vez um sistema que se apoia de maneira importante na noção de computação indutiva.

Todas essas observações parecem indicar que a exploração de estruturas indutivas são um caminho natural para implementar a automação em assistentes de prova. Isso faz sentido, pois a estrutura indutiva fornece uma forma de direcionamento para a aplicação dos fatos matemáticos, o que permite instanciar a noção de computação. As observações do parágrafo anterior também mostram que essa forma particular de computação pode ser acoplada ao processo de conversão para a obtenção de uma série de benefícios, como a elisão de provas e a integração da computação

à parte lógica do sistema. Mais especificamente, se o assistente de prova é capaz de verificar sozinho a equivalência entre dois tipos (via computação), então não é preciso incluir nenhuma justificativa desse fato no objeto de prova, e é isso o que nós chamamos de elisão de provas. E, se o assistente de prova tem essa capacidade computacional, então certos passos lógicos do sistema podem ser realizados via computação.

Portanto, a resposta que nós queremos oferecer para a pergunta acima é: Basta seguir o caminho apontado por estruturas indutivas. De fato, é exatamente isso o que acontece no CIC. As estruturas indutivas são introduzidas no cálculo por meio de definições de tipos indutivos. Por um lado, essa definição especifica os construtores do tipo, e por outro lado, ela introduz automaticamente no contexto o eliminador associado ao tipo. O eliminador, por sua vez, permite formular definições indutivas de funções que operam sobre objetos do tipo indutivo. Essas definições essencialmente consistem de regras que especificam o comportamento da função para cada construtor. Dessa maneira, o cálculo do valor da função sobre um termo formado pela aplicação de construtores consiste em utilizar a regra correspondente ao construtor mais externo, sucessivamente. Finalmente, como em cada passo só existe uma regra disponível, esse processo pode ser realizado de maneira automática. Ou seja, em um certo sentido, a estrutura sintática do termo direciona o conteúdo matemático da definição, de forma que ele pode ser interpretado como um conjunto de regras computacionais.

Mas, ao associar o mecanismo de computação às definições indutivas, o CIC acaba impondo restrições excessivas à computação que está disponível no sistema. Mais especificamente, as regras computacionais devem satisfazer as mesmas condições que são exigidas da definição. Por exemplo, as regras de uma definição indutiva devem ser exaustivas e mutuamente exclusivas, isto é, deve existir um único argumento que controla a recursão, e exatamente uma regra para cada construtor do tipo desse argumento. Condições desse tipo desempenham um papel importante nas definições indutivas, pois elas garantem que a definição é completa e não é ambígua. Mas, o exemplo a seguir mostra como essas condições podem restringir a computação. A definição típica de soma sobre números naturais no CIC nos dá as regras de computação por recursão à esquerda $0 + x \to x$ e $(S\,x) + y \to S\,(x + y)$. Dessa maneira, um termo como $1 + x \equiv (S\,0) + x$ reduz em dois passos para $S\,x$. Por outro lado, o termo $x + 1 \equiv x + (S\,0)$ é irredutível. Agora, suponha que as regras que correspondem à recursão à direita, $x + 0 \to x$ e $x + (S\,y) \to S(x + y)$, também tem caráter computacional. Nesse caso, o termo $x + 1$ também reduz em dois passos para $S\,x$. No entanto, essa suposição viola a condição mencionada acima,

pois as quatro regras definem uma recursão sobre dois argumentos. Ou seja, a computação associada à definição típica da soma é mais restrita do que é possível obter. A partir dessa observação, abrem-se dois caminhos para tentar introduzir mais computação ao sistema. O primeiro consiste em utilizar esquemas de definição mais gerais (e.g. definição por regras de reescrita (CHRZĄSZCZ; WALUKIEWICZ-CHRZĄSZCZ, 2007)). Em princípio isso pode funcionar, mas ao abandonar a noção de definições indutivas, nós perdemos de vista a ideia obtida acima de que é natural implementar automação ao longo de estruturas indutivas. Por essa razão, nós vamos adotar o segundo caminho, que consiste em desacoplar a computação das definições indutivas (funções) – mantendo a computação associada a estruturas indutivas!

Para fazer isso, nós vamos resgatar a nossa intuição original de transformar conteúdo matemático, mais especificamente teoremas, em computação. Em outras palavras, a nossa proposta consiste em eliminar a computação presente nas definições indutivas e recuperar esta computação através de teoremas. Nesse sentido, nós queremos utilizar enunciados de teoremas como regras de computação, de modo que eles desempenhem um papel semelhante ao que as regras das definições indutivas tem no CIC. O único problema é que, em geral, teoremas não são direcionados. Um caso típico são teoremas que estabelecem a igualdade entre dois termos, que pode ser utilizada nas duas direções. A nossa ideia chave, então, consiste em utilizar estruturas indutivas para indicar a direção em que o teorema deve ser utilizado em uma computação. Mais especificamente, se os termos que aparecem no teorema já possuem um tipo indutivo, então basta examinar se uma das direções do teorema pode ser interpretada como uma regra de redução ao longo dessa estrutura indutiva. Por exemplo, o teorema que estabelece a relação de recursão à direita na função de soma, $\Pi x, y.S(x+y) = x+(Sy)$, pode ser direcionado para obter a regra $x+(Sy) \to S(x+y)$. E, quando isso não é o caso, pode ser possível explicitar a estrutura indutiva dos termos a posteriori, identificando subtermos que fazem o papel de construtores, de modo que o teorema possa ser interpretado como uma regra de redução ao longo dessa estrutura. Por exemplo, considere uma definição não-indutiva dos números naturais e das funções de soma e multiplicação (e.g. via a noção de semianel inicial). Nesse caso, os termos que representam números naturais não são formados por construtores, estritamente falando. Por outro lado, nós podemos interpretar termos da forma $1+x$ como o sucessor de x, de modo que o subtermo $1+$ cumpre o papel do construtor S. Uma vez que essa observação foi feita, nós podemos direcionar o teorema $\Pi x.1+x = x+1$ para obter a regra de computação $x+1 \to 1+x$. Utilizando a mesma ideia, nós também podemos direcionar o teorema $\Pi xy, x*(1+y) = x+(x*y)$ para obter a regra

de computação $x * (1 + y) \rightarrow x + (x * y)$. Agora, não é difícil ver que o termo $x * (y + 1)$ reduz em dois passos para $x + (x * y)$. Esses exemplos mostram que a nossa solução permite obter, no primeiro caso, mais computação do que se tem no CIC, e no segundo caso, obter computação em uma situação em que ela não existe no CIC. Mas, é preciso fazer isto com cuidado, pois existem condições que devem ser satisfeitas para que a computação esteja bem-definida. Por exemplo, o conjunto de regras obtidos a partir do direcionamento de teoremas pode gerar uma computação não-terminante. Na seção seguinte, nós vamos discutir esses e outros aspectos técnicos associados à nossa proposta.

6.1 Aspectos técnicos

Como mencionamos acima, a nossa ideia consiste em desacoplar a computação das definições e recuperá-la por meio de um mecanismo que utiliza teoremas como regras de computação. E a chave para fazer isto consiste em lançar mão da seguinte observação feita no sistema VeriML(STAMPOULIS, 2013): O mecanismo de conversão é essencialmente uma tática em que o sistema confia [1]. Logo, a nossa proposta pode ser vista como a ideia de trocar o mecanismo de conversão presente em um sistema como o CC ou CIC, por um mecanismo de conversão que utiliza teoremas como regra de redução. É claro que isso precisa ser feito preservando a confiança do sistema no mecanismo de conversão. Existem ao menos duas maneiras de conseguir isto. A primeira delas consiste em investigar a natureza da computação que está sendo introduzida para garantir as suas boas propriedades. E a segunda delas consiste em utilizar uma solução que já fez uma investigação similar, como o sistema VeriML. O VeriML consiste em duas partes: um sistema formal, que é muito similar ao CC mas não possui a regra de conversão, e uma linguagem de programação, que permite a construção de táticas (isto é, programas que geram termos do sistema formal) e utiliza a primeira parte como sistema de tipos. Dessa maneira, quando as táticas possuem um tipo correto, elas correspondem a uma computação confiável. Como o foco desse trabalho é entender o papel da computação em um assistente de prova e investigar a melhor maneira de acoplá-la ao sistema, a seguir nós vamos nos restringir a descrever o novo mecanismo de conversão, e assumir que uma eventual implementação do sistema seguirá por um desses caminhos.

[1] A passagem específica é a seguinte: "The crucial step is to recognize that the conversion rule essentially consists of a trusted tactic that is hardcoded within the logic type checker. This tactic checks whether two terms are definitionally equal or not; if it claims that they are, we trust that they are indeed so, and no proof object is produced."(STAMPOULIS, 2013, p.156)

O mecanismo de conversão presente no CIC é extremamente simples: após aplicar sucessivas reduções aos dois termos para obter as correspondentes formas normais, o sistema verifica se elas são sintaticamente iguais. A nossa modificação consiste apenas em alterar o conjunto de regras de redução que é utilizado nesse mecanismo. Mas, como já sabemos, é preciso fazer isso com cuidado, pois o conjunto de regras de redução precisa definir uma computação com boas propriedades. Por exemplo, imagine que as regras de redução $x + (Sy) \to (Sx) + y$ e $(Sx) + y \to x + (Sy)$ estão presentes no conjunto. Nesse caso, é fácil ver que a aplicação sucessiva dessas regras pode gerar um laço infinito. Aqui mais uma vez existem ao menos dois caminhos possíveis para certificar o conjunto de regras de redução. O primeiro consiste em investigar em que condições esse conjunto tem as propriedades desejadas (e.g. normalização forte, consistência lógica, etc). Por outro lado, a questão sobre regras de reescrita definirem uma computação com boas propriedades já foi investigada extensivamente na literatura. O segundo caminho, portanto, seria adotar algum critério como General Scheme(BLANQUI, 2001) e HORPO(WALUKIEWICZ-CHRZĄSZCZ, 2003) que certifica propriedades específicas de um conjunto de regras de reescrita, como a terminação da computação. O problema da consistência lógica da integração desses critérios a um assistente de provas como o Coq é considerado em (CHRZĄSZCZ; WALUKIEWICZ-CHRZĄSZCZ, 2007). Como fizemos acima, nós vamos assumir que uma eventual implementação do sistema irá escolher um desses caminhos para certificar o conjunto de reduções.

Finalmente, as regras de redução que nós pretendemos utilizar serão obtidas com o direcionamento de teoremas baseado em alguma estrutura indutiva. Essa restrição adicional introduz duas possibilidades interessantes. A primeira delas é a ideia de obter a propriedade de consistência lógica imediatamente a partir do fato de que a computação é baseada em teoremas. E a segunda é a possibilidade de obter um critério mais abrangente do que HORPO e General Scheme, a partir do fato de que as regras de redução são orientadas por meio de uma estrutura indutiva [2]. Por outro lado, as garantias mencionadas nos parágrafos anteriores já são suficientes, e nós podemos continuar a descrever a nossa proposta.

O elemento central da nossa proposta é a ideia de explorar a computação ao longo de estruturas indutivas. Considere primeiramente o caso em que os tipos foram definidos indutivamente. Já existe, nesse caso, uma estrutura indutiva descrita no sistema que pode ser utilizada para orientar os teoremas. Mas uma das consequências mais interessantes do

[2] Estas questões correspondem a um passo adiante interessante, e nós pretendemos nos dedicar a elas em trabalhos futuros.

desacoplamento entre a computação e as definições é que a ideia de que a computação pode ser explorada ao longo de estruturas indutivas que não correspondem à definição básica dos termos. Mais especificamente, nós podemos projetar uma segunda estrutura indutiva sobre o tipo, explicitando expressões que fazem o papel de construtores. Por exemplo, considere a definição típica do CIC para os números naturais, que é formulada em termos de uma estrutura indutiva unária. Então, nós podemos identificar as expressões 0, $(2 * x) + 1$ e $(2 * x) + 2$ desse tipo como construtores de uma estrutura indutiva binária para os números naturais. A ideia é que essa estrutura indutiva alternativa pode ser explicitada para o sistema, para orientar teoremas e obter regras de redução. Agora, considere o caso em que os tipos não foram definidos indutivamente. Aqui, não há realmente alternativa, pois para obter qualquer tipo de computação é preciso projetar uma estrutura indutiva sobre o tipo. Um exemplo deste tipo foi apresentado no final da seção anterior. Em todos os casos, o que caracteriza a estrutura indutiva é a *gramática de construtores* que define os termos do tipo indutivo, e é ela que determina a direção em que os teoremas serão orientados.

Do ponto de vista operacional, nós imaginamos que o usuário é responsável por definir as estruturas indutivas que serão utilizadas, mais especificamente as suas gramáticas de construtores, e indicar explicitamente os teoremas que ele deseja direcionar. As regras de redução assim obtidas são registradas no sistema para uso posterior no mecanismo de conversão. Começando em um sistema como o CIC, o primeiro passo consiste em desabilitar a computação automática associada às definições de tipos indutivos. Ao fazer isso, no entanto, surge o problema de que certos teoremas, que só podem ser demonstrados por computação no CIC, não podem mais ser provados. Para resolver essa dificuldade, nós utilizamos mais uma ideia encontrada em (STAMPOULIS, 2013): introduzir igualdades explícitas no sistema, que correspondem às ι-reduções. Esse passo basicamente nos dá uma versão do CIC sem computação indutiva. O passo seguinte consiste em demonstrar teoremas e orientá-los com base em estruturas indutivas para obter regras de redução. Abaixo nós temos alguns exemplos que ilustram o funcionamento desse mecanismo:

a) Esse primeiro exemplo mostra que é imediato recuperar a computação presente no CIC. A partir da definição

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat
end.
```

nós obtemos o tipo *nat*, tal como vimos no capítulo 4, que define uma gramática para termos do tipo *nat*. Esta definição também inclui implicitamente o indutor *nat_ind* e as novas igualdades que correspondem à ι-redução sobre o indutor. A seguir, a função soma é definida utilizando o indutor como:

```
Definition plus n m := nat_ind (fun x => nat) n (
   fun n r => S r) m.
```

Finalmente, nós podemos enunciar os teoremas:

```
Theorem plus_O_r : forall n, plus n O = n.
Theorem plus_S_r : forall n m, plus n (S m) = S (
   plus n m).
```

que podem ser demonstrados trivialmente utilizando as igualdades explícitas introduzidas pela definição do tipo *nat*. Agora, basta observar que os teoremas podem ser orientados para registrar as seguintes regras de redução:

```
plus n O -> n.
plus n (S m) -> S (plus n m).
```

Nós estamos assumindo que essas regras são armazenadas em estruturas de dados que são consultadas pelo mecanismo de conversão. Dessa maneira, a computação associada à soma no CIC é recuperada no nosso sistema.

b) De fato, existe uma maneira mais interessante e eficiente de recuperar toda a computação associada ao tipo *nat* de uma só vez. Note que as igualdades explícitas que correspondem a ι-redução

$$nat\_ind\_O : \Pi P, PO, PS.\,(nat\_ind\ P\ PO\ PS\ 0 = PO)$$

$$nat\_ind\_S : \Pi P, PO, PS, m.\,(nat\_ind\ P\ PO\ PS\ (S\ m) = PS\ m\ (nat\_ind\ P\ PO\ PS\ m))$$

que podem ser interpretadas como teoremas, e orientadas para obter as regras de redução:

```
nat_ind P PO PS O     -> PO
nat_ind P PO PS (S m) -> PS m (nat_ind P PO PS m)
```

Isso significa que qualquer definição formulada em termos de *nat_ind*, como a definição de soma no item anterior, herda a computação associada ao indutor. Dessa maneira, a computação associada ao tipo *nat* é recuperada através do mesmo mecanismo que é implementado no CIC (i.e. associando computação ao indutor).

c) O nosso próximo exemplo mostra como obter mais computação que no CIC. Não é difícil ver que os teoremas abaixo seguem com facilidade da função soma:

```
1 Theorem plus_O_l : forall m, plus O m = m.
2 Theorem plus_S_l : forall n m, plus (S n) m = S (
     plus n m).
```

E eles podem ser orientados para obter as regras de redução

```
plus O m -> n.
plus (S n) m -> S (plus n m).
```

que correspondem à recursão à esquerda. Essas regras devem funcionar juntamente com as regras de recursão à direita obtidas na definição, assumindo que o conjunto como um todo possui as boas propriedades que garantem uma computação segura (e.g. satisfazendo HORPO). Dessa maneira, nós conseguimos mais computação associada a função de soma do que se tem no CIC.

d) Outra maneira de conseguir mais computação do que no CIC consiste em utilizar múltiplas estruturas indutivas associadas ao mesmo tipo. Por exemplo, o tipo *nat* definido com uma estrutura indutiva de construtores *O* e *S*, também pode ser descrito pela estrutura estrutura binária abaixo:

```
Inductive Grammar nat_pos : nat :=
| 1 : nat
| (2*_) : nat -> nat
| (2*_ + 1) : nat -> nat.
```

Note que nós estamos definindo uma gramática de construtores alternativa para o tipo original. De fato, essa gramática corresponde apenas aos números inteiros positivos, mas isso não é um problema, pois ela será utilizada apenas para fins computacionais. Agora, imagine que as funções *parity*, *div*2 e *log*2 já foram definidas, e os seguintes teoremas já foram provados:

```
Theorem parity_twox : forall n, parity (2*n) =
   true
Theorem div2_twox : forall n, div2 (2*n) = n.
Theorem log2_twox : forall n, log2 (2*n) = S (
   log2 n).
```

Então, os teoremas podem ser orientados para obter as regras de redução

```
parity (2*n) -> true
div2 (2*n) -> n
log2 (2*n) -> S (log2 n)
```

e) Finalmente, suponha que os números naturais foram definidos via a noção de semianel inicial. Nesse caso, nós podemos projetar a estrutura indutiva descrita pela gramática abaixo sobre esse tipo.

```
Inductive Grammar N_unary : N_semiring :=
| 0 : N_semiring
| (1 + _) : N_semiring -> N_semiring.
```

A seguir, teoremas como

```
1 Theorem plus_O_nsr : forall (n:N_semiring), n+0
    = n
2 Theorem plus_S_nsr : forall (n m:N_semiring), n
    +(1+m) = 1+(n+m)
```

podem ser orientados para obter as regras de redução

```
n+0 -> n
n+(1+m) -> 1+(n+m)
```

Dessa maneira, nós obtermos computação sobre uma definição que não pode ser automatizada no CIC.

REFERÊNCIAS

BARENDREGT, H. Introduction to generalized type systems. **Journal of functional programming**, Cambridge University Press, v. 1, n. 2, p. 125–154, 1991.

BARENDREGT, H.; COHEN, A. M. Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. **Journal of Symbolic Computation**, Elsevier, v. 32, n. 1-2, p. 3–22, 2001.

BARRAS, B.; JOUANNAUD, J.-P.; STRUB, P.-Y.; WANG, Q. Coqmtu: a higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In: IEEE. **Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on**. [S.l.], 2011. p. 143–151.

BLANQUI, F. **Théorie des types et réécriture**. Tese (Doutorado) — Université Paris Sud-Paris XI, 2001.

BLANQUI, F.; JOUANNAUD, J.-P.; STRUB, P.-Y. Building decision procedures in the calculus of inductive constructions. In: SPRINGER. **International Workshop on Computer Science Logic**. [S.l.], 2007. p. 328–342.

BÖHM, C.; BERARDUCCI, A. Automatic synthesis of typed λ-programs on term algebras. **Theoretical Computer Science**, Elsevier, v. 39, p. 135–154, 1985.

BOUTIN, S. Using reflection to build efficient and certified decision procedures. In: SPRINGER. **International Symposium on Theoretical Aspects of Computer Software**. [S.l.], 1997. p. 515–529.

CHLIPALA, A. **Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant**. [S.l.]: MIT Press, 2013.

CHRZĄSZCZ, J.; WALUKIEWICZ-CHRZĄSZCZ, D. Towards rewriting in coq. In: **Rewriting, Computation and Proof**. [S.l.]: Springer, 2007. p. 113–131.

COQUAND, T. **Metamathematical investigations of a calculus of constructions**. Tese (Doutorado) — INRIA, 1989.

COQUAND, T.; HUET, G. The calculus of constructions. **Information and computation**, Elsevier, v. 76, n. 2-3, p. 95–120, 1988.

FIRSOV, D.; STUMP, A. Generic derivation of induction for impredicative encodings in cedille. **Certified Programs and Proofs (CPP)**, 2018.

GEUVERS, H. Induction is not derivable in second order dependent type theory. In: SPRINGER. **International Conference on Typed Lambda Calculi and Applications**. [S.l.], 2001. p. 166–181.

GONTHIER, G.; ASPERTI, A.; AVIGAD, J.; BERTOT, Y.; COHEN, C.; GARILLOT, F.; ROUX, S. L.; MAHBOUBI, A.; O'CONNOR, R.; BIHA, S. O. *et al.* A machine-checked proof of the odd order theorem. In: SPRINGER. **International Conference on Interactive Theorem Proving**. [S.l.], 2013. p. 163–179.

GONTHIER, G.; MAHBOUBI, A.; TASSI, E. **A Small Scale Reflection Extension for the Coq system**. [S.l.], 2016. Disponível em: <https://hal.inria.fr/inria-00258384>.

HEDBERG, M. A coherence theorem for martin-löf's type theory. **Journal of Functional Programming**, Cambridge University Press, v. 8, n. 4, p. 413–436, 1998.

JOUANNAUD, J.-P.; STRUB, P.-Y. Coq without type casts: A complete proof of coq modulo theory. In: **LPAR-21: 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning**. [S.l.: s.n.], 2017.

KAUFMANN, M.; MOORE, J. S. An industrial strength theorem prover for a logic based on common lisp. **IEEE Transactions on Software Engineering**, IEEE, v. 23, n. 4, p. 203–213, 1997.

MATUSZEWSKI, R.; RUDNICKI, P. Mizar: the first 30 years. **Mechanized mathematics and its applications**, Citeseer, v. 4, n. 1, p. 3–24, 2005.

MULLIGAN), D. M. (https://cstheory.stackexchange.com/users/375/dominic. **How do 'tactics' work in proof assistants?** 2011. Theoretical Computer Science Stack Exchange. URL:https://cstheory.stackexchange.com/q/8508 (version: 2011-10-07). Disponível em: <https://cstheory.stackexchange.com/q/8508>.

PAULIN-MOHRING, C. Inductive definitions in the system coq rules and properties. In: SPRINGER. **International Conference on Typed Lambda Calculi and Applications**. [S.l.], 1993. p. 328–345.

PAULIN-MOHRING, C. **Introduction to the calculus of inductive constructions**. [S.l.]: College Publications, 2015.

PFENNING, F.; PAULIN-MOHRING, C. Inductively defined types in the calculus of constructions. In: SPRINGER. **International Conference on Mathematical Foundations of Programming Semantics**. [S.l.], 1989. p. 209–228.

POINCARÉ, H. La science et l'hypothèse. **La Science et**, v. 1, 1902.

PUGH, W. The omega test: a fast and practical integer programming algorithm for dependence analysis. In: ACM. **Proceedings of the 1991 ACM/IEEE conference on Supercomputing**. [S.l.], 1991. p. 4–13.

SHOSTAK, R. E. A practical decision procedure for arithmetic with function symbols. **Journal of the ACM (JACM)**, ACM, v. 26, n. 2, p. 351–360, 1979.

SOZEAU, M.; TABAREAU, N. Universe polymorphism in coq. In: SPRINGER. **International Conference on Interactive Theorem Proving**. [S.l.], 2014. p. 499–514.

STAMPOULIS, A. M. **VeriML: A Dependently-typed, user extensible and language-centric approach to proof assistants**. [S.l.]: Yale University, 2013.

STRUB, P.-Y. **Type Theory and Decision Procedures**. Tese (Doutorado) — Ecole Polytechnique X, 2008.

STRUB, P.-Y. Coq modulo theory. In: SPRINGER. **International Workshop on Computer Science Logic**. [S.l.], 2010. p. 529–543.

THE COQ DEVELOPMENT TEAM. **The Coq proof assistant reference manual**. [S.l.], 2017. Version 8.7. Disponível em: <http://coq.inria.fr>.

WALUKIEWICZ-CHRZĄSZCZ, D. Termination of rewriting in the calculus of constructions. **Journal of Functional Programming**, Cambridge University Press, v. 13, n. 2, p. 339–414, 2003.