UNIVERSIDADE DE BRASÍLIA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE MATEMÁTICA

Uma Formalização da Teoria de Reescrita em Linguagem de Ordem Superior

Por

André Luiz Galdino

Brasília
2008

UNIVERSIDADE DE BRASÍLIA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE MATEMÁTICA

Uma Formalização da Teoria de Reescrita em Linguagem de Ordem Superior

Por

André Luiz Galdino[1]

Orientador: Prof. Dr. Mauricio Ayala Rincón

[1]O autor contou com o apoio financeiro parcial do CNPq.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Matemática

Uma Formalização da Teoria de Reescrita em Linguagem de Ordem Superior

por

André Luiz Galdino *

Tese apresentada ao Departamento de Matemática da Universidade de Brasília, como parte dos requisitos para obtenção do grau de

Doutor em Matemática

29 de agosto de 2008

Comissão Examinadora:

Prof. Maurício Ayala Rincón - MAT/UnB (Orientador)

Profa. Elaine Gouvêa Pimentel - MAT/UFMG (Membro)

Prof. Edward Hermann Haeusler - PUC/Rio (Membro)

Prof. Marcelo Finger - IME/USP (Membro)

Prof. Cesar Muñoz - National Institute of Aerospace - NIA/NASA (Membro)

*O autor contou com o apoio financeiro parcial do CNPq.

Dedico este a meus pais Maria Aparecida e Antero Galdino,
a meus irmãos Paulo Henrique e Dilourdes Maria,
a minha esposa Rosirene Santana
e aos meus filhos Rafael e Iury.

"Deus, concede-me serenidade para aceitar as coisas que não posso mudar.
Coragem para mudar as coisas que eu posso.
Sabedoria para saber diferenciar entre as duas coisas,
e que eu seja razoavelmente feliz nessa vida."

Agradecimentos

Num primeiro momento quero agradecer a Deus, por ter dado-me forças para que eu pudesse finalizar este trabalho, aos meus pais e meus irmãos que me incentivaram e apoiaram durante toda minha jornada de estudos.

Por incetivar-me nos momentos de indecisão, pelo apoio e compreensão nesta difícil caminhada agradeço, com todo carinho, a minha esposa e filhos.

Quero agradecer a meu orientador, Prof. Mauricio Ayala Rincón, que além de ser um ótimo orientador, foi conselheiro e amigo, e sem suas valiosas orientações não seria possível a concretização deste trabalho.

Meus agradecimentos ao CNPq, Prefeitura Municipal de Catalão, FINATEC e FUNAPE pelos apoios financeiros, parciais, os quais possibilitaram e incentivaram a conclusão deste trabalho.

A Porfírio Azevedo e Élida Alves pela amizade, paciência e incentivo em todos os momentos da minha caminhada meus eternos agradecimentos. Aproveito para agradecer a todos os professores do Departamento de Matemática do *Campus* Catalão da Universidade Federal de Goiás que direto ou indiretamente contribuíram para o término deste trabalho.

Pelas conversas, em horas de dúvidas, pelo apoio e incentivo, quero agradecer a Aline de Souza, Daniel Ventura, Fernando Kennedy, Flávio Leonardo, Jhone Caldeira, Plínio José, Thiago Porto e demais colegas que tiveram "paciência" para comigo durante todo o curso.

Finalmente, agradeço a todos os meus amigos, colegas, professores e funcionários do Departamento de Matemática da UnB que direto ou indiretamente contribuíram para o término deste trabalho.

Resumo

Teorias para Sistemas Abstratos de Redução (ARS) e Sistemas de Reescrita de Termos (TRS) no assistente de provas PVS (Prototype Verification System) chamadas `ars` e `trs`, respectivamente, foram desenvolvidas. A *teoria* `ars`, construída com base na teoria para relações binárias do PVS, contém especificações de noções tais como redução, confluência, formas normais, e conceitos não básicos como por exemplo noeterianidade. Por outro lado, a *teoria* `trs`, construída com base na *teoria* `ars` e a teoria para seqüências finitas encontrada na biblioteca do PVS, contém uma formalização para lidar com a estrutura dos termos, assim como, formalizações de noções não triviais de TRS. As *teorias* `ars` e `trs` foram desenvolvidas com o objetivo de agregar os conceitos e as definições necessários para lidar com a Teoria de Reescrita, em geral. Em outras palavras, `ars` e `trs` contém elementos que formam uma base sólida para formalizar propriedades da Teoria de Reescrita em PVS. Para certificar-se de que o objetivo foi alcançado vários resultados bem conhecidos e não triviais foram formalizados; dentre estes, destacam-se a correção do princípio de indução Noeteriana, o Lema de Newman, os Lemas de Comutação e o Teorema dos Pares Críticos de Knuth-Bendix. Além de constituir uma base para formalização de propriedades da Teoria de Reescrita, em geral, a formalização apresentada se destaca por: 1. utilizar uma linguagem de orderm superior, a qual permite expressar naturalmente propriedades de ordem superior; 2. por seu alto grau de abstração, que permite expressar propriedades numa forma quasi-geométrica, como desejável em Teoria de Reescrita; e, 3. pelo alto grau de controle, permitido pelo PVS, no desenvolvimento das provas.

Abstract

Theories for Abstract Reduction Systems (ARS) and Term Rewriting Systems (TRS) in the proof assistant PVS (Prototype Verification System) called `ars` and `trs`, respectively, we developed. The `ars` *theory* built on the PVS library for binary relations, contains specifications of notions such as reduction, confluence, normal forms, and non basic concepts such as Noetherianity. On the other hand, the `trs theory` built on the `ars theory` and the PVS library for finite sequences, contains a formalization to deal with the structure of terms as well as formalizations of non-trivial notions of TRS. Theories `ars` and `trs` were developed with the main goal of providing the necessary concepts and definitions to deal with the Theory of Rewriting in general. In other words, `ars` and `trs` contain elements that conform a solid basis to formalize properties of the Theory of Rewriting in PVS. To make sure that the goal was achieved well-known and non-trivial results were formalised; among these, the correctness of the principle of noetherian induction, the Newman's Lemma, the Commutation Lemma and the Knuth-Bendix Critical Pair Theorem. Apart from being a basis for formalization of properties of the Theory of Rewriting, in general, the formalization presented is highlighted by: 1. the use a higher-order language, which allows for the specification of high-order properties naturally, 2. for their high-level of abstraction, which allows for the specification properties in an almost geometric style, as desirable in Rewriting Theory, and 3. the high degree of control allowed by PVS in the development of proofs.

Índice

Capítulo 1

Introdução

Esta introdução esta organizada em seções da seguinte forma: A primeira seção apresenta uma motivação sobre a Teoria de Reescrita. A segunda seção apresenta a contribuição deste trabalho. A terceira seção apresenta alguns trabalhos relacionados, e a quarta e última seção apresenta a organização geral deste trabalho.

1.1 Motivação: Teoria da Reescrita

É um fato que as *Equações*, que são igualdades envolvendo expressões matemáticas, ocupam um lugar de destaque dentro da Matemática e de suas aplicações. Por exemplo, o estudo de equações algébricas levou ao surgimento da Teoria dos grupos, Teoria dos Corpos, etc, ou seja, ao surgimento da Álgebra, já as equações diferenciais e integrais foram a base de construção da Análise Matemática, e, do ponto de vista computacional, a Análise Numérica nada mais é do que métodos ou algoritmos para resolver problemas equacionais de difícil tratamento com papel e lápis.

Em geral, sempre nos deparamos com situações onde desejamos determinar se uma identidade é válida, ou buscar soluções para uma equação, ou simplesmente simplificar uma dada equação. A habilidade em resolver estes problemas equacionais dá suporte para entender e lidar com várias aplicações computacionais, tais como: ambientes de especificação e verificação, e linguagens de programação de ordem-superior. Neste sentido, uma das áreas de estudo na Teoria da Computação que vem conquistando espaço é a verificação (semi)automática de igualdade entre expressões, com aplicações imediatas em

problemas como prova (semi)automática de teoremas, programação em lógica e verificação de consistência na especificação de tipos abstratos de dados. Esse interesse pelo tratamento (semi)automático de problemas da matemática, como a simplificação de expressões e demonstração de teoremas, surgiu antes mesmo do surgimento de uma tecnologia que possibilitasse a implementação em máquina de tais idéias.

Por exemplo, pela aritmética usual dos números inteiros, a expressão $(-(-a)+(-a))+a$ tem o mesmo significado que as expressões $0+a$ e a e é comum, neste caso, dizer que estas expressões são iguais e escreve-se usualmente $(-(-a)+(-a))+a = 0+a$, $0+a = a$, etc. Aqui usaremos $\approx$ em vez de $=$, por exemplo, $0+a \approx a$. Assim, podemos garantir que $0+a \approx a$ devido ao fato de que estamos assumindo, como verdade pré-estabelecida, que toda expressão da forma $0+x$ é igual a x, onde x é uma variável, ou seja, $0+x \approx x$. Logo, substituindo a no lugar de x obtemos $0+a \approx a$. De fato, a aritmética usual dos inteiros satisfaz a estrutura algébrica usual de grupos, onde são válidos os seguintes axiomas equacionais:

$$\begin{array}{lrcl} \text{(A1)} & 0+x & \approx & x \\ \text{(A2)} & -x+x & \approx & 0 \\ \text{(A3)} & x+(y+z) & \approx & (x+y)+z \end{array}$$

Neste sistema de axiomas equacionais temos, por exemplo, $(-(-a)+(-a))+a \approx^{(A2)} 0+a \approx^{(A1)} a$. Observe que a expressão a, resultante das aplicações dos axiomas dados acima, é justamente aquela menos complexa (em outras palavras, a mais curta) dentre todas as expressões que possuem o mesmo significado da expressão $(-(-a)+(-a))+a$ inicialmente considerada.

Vamos supor agora que desejamos implementar em um ambiente computacional um algoritmo que faça simplificações (*reduções*), como a apresentada acima, respeitando um conjunto de axiomas equacionais. É natural, e interessante, almejar que tal algoritmo seja de tal forma que possamos ter como garantia que toda sequência de aplicações dos axiomas realizada por ele seja finita (ou seja, que tenha a *propriedade noeteriana*). Porém, vale observar que tal comportamento não é natural, tampouco trivial, pois, a simples aplicação dos axiomas equacionais pode não garantir que toda sequência de redução seja finita. Por exemplo, com o sistema de axiomas equacionais exposto acima, podemos fazer sucessivas

aplicações dos axiomas $(A1)$ e $(A2)$ sobre a expressão $(-(-a) + (-a)) + a$ considerada inicialmente:

$$(-(-a) + (-a)) + a \approx^{(A2)} 0 + a \approx^{(A1)} a \approx^{(A1)} 0 + a \approx^{(A2)} (-(-a) + (-a)) + a \cdots$$

isto acontece porque, usualmente, do ponto de vista algébrico é natural considerar que os axiomas equacionais são simétricos, ou seja, a ordem com que são aplicados é considerada irrelevante, em outras palavras, podem ser usados em ambas direções.

Assim, sob a motivação de encontrar um método para simplificar expressões de modo (semi)automático e tratar computacionalmente as equações (axiomas equacionais) surge, inicialmente com o objetivo de formalizar o conceito de funções computáveis, um poderoso método conhecido como *Sistemas de Reescrita de Termos* [4, 6]. Um exemplo importante de aplicação de Sistemas de Reescrita de Termos é para resolver o *Problema da Palavra* [43] que é: *dado um conjunto de equações, decidir se dois termos são provavelmente iguais.*

Nestes sistemas, as equações orientadas, chamadas *regras de reescrita* as quais são geradas inicialmente a partir do conjunto de axiomas equacionais considerado, são usadas para repassar iguais (*termos*) por iguais (*termos*) em uma única direção. A idéia de orientar as equações, geralmente da "esquerda para a direita", é uma tentativa de garantir que toda sequência de redução seja finita. Em outras palavras, tentando garantir que quando um regra de reescrita é aplicada a um termo t_0 então o termo resultante t_1 seja menos complexo do que o termo t_0, ou seja, que exista uma relação $\prec$ bem-fundada (noeteriana) sobre os termos. Como já citamos antes, nem sempre é uma tarefa fácil encontrar uma tal relação de ordem. Em certos casos encontrar tal relação se torna impossível, por exemplo, se a estrutura algébrica considerada for comutativa.

Nos Sistemas de Reescrita de Termos, interpretamos as identidades como regras de reescrita que nos dizem como um subtermo de um dado termo deve ser repassado por um outro termo. Entretanto, quando usamos (aplicamos) uma regra de reescrita, a usamos em uma única direção e, para expressarmos esta idéia, usamos o símbolo $x \rightarrow y$ em vez de $x \approx y$, e dizemos que x *reduz para* y. Para um melhor entendimento consideremos o sistema de reescrita de termos abaixo, obtido do sistema de axiomas equacionais para estruturas algébricas de grupos apresentado anteriormente, acrescido do axioma $x + y \approx$

$y + x$, o qual expressa a comutatividade da adição para os inteiros:

$$\begin{array}{lrcl} (\text{A1}) & 0 + x & \to & x \\ (\text{A2}) & -x + x & \to & 0 \\ (\text{A3}) & x + (y + z) & \to & (x + y) + z \\ (\text{A4}) & x + y & \to & y + x \end{array}$$

A sequência $a + b \to^{(A4)} b + a \to^{(A4)} a + b \to^{(A4)} \cdots$ é um exemplo de uma cadeia infinita de aplicações da regra $(A4)$.

Vejamos mais alguns exemplos:

Soma de Naturais: Vamos definir a adição de números naturais, usando a constante 0 e a função sucessor s, com as seguintes identidades:

$$\begin{array}{rcl} x + 0 & \approx & x \\ x + s(y) & \approx & s(x + y) \end{array}$$

Podemos aplicar estas identidades e calcular a soma de 1 e 2, que são representados por $s(0)$ e $s(s(0))$, respectivamente:

$$s(0) + s(s(0)) \approx s(s(0) + s(0)) \approx s(s(s(0) + 0)) \approx s(s(s(0))).$$

Assim, o sistema de axiomas equacionais mostrado acima, pode ser apresentado em forma de um Sistema de Reescrita de Termos como segue:

$$\begin{array}{lrcl} (\text{a}) & x + 0 & \to & x \\ (\text{b}) & x + s(y) & \to & s(x + y) \end{array}$$

e consequentemente temos,

$$s(0) + s(s(0)) \to^{b} s(s(0) + s(0)) \to^{b} s(s(s(0) + 0)) \to^{a} s(s(s(0))).$$

onde $\to^{\square}$ indica qual regra de reescrita estamos aplicando. Neste exemplo $\square = a$ ou b.

Fatorial: Consideremos o seguinte Sistema de Reescrita para computar a função fatorial:

$$\begin{array}{llcllcl} (1) & 0 + x & \to & x & (4) & s(x) + y & \to & s(x + y) \\ (2) & 0 \times x & \to & 0 & (5) & s(x) \times y & \to & y + (x \times y) \\ (3) & fact(0) & \to & s(0) & (6) & fact(s(x)) & \to & s(x) \times fact(x) \end{array}$$

Em ambos exemplos acima, $+$ e $\times$ são símbolos de funções binárias, s e $fact$ são símbolos de funções unárias, 0 é uma constante e, x e y são variáveis.

Com base no último Sistema de Reescrita vamos computar $fact(2)$ onde 2 é representado por $s(s(0))$:

$$\begin{aligned}
fact(s(s(0))) &\to^6 s(s(0)) \times fact(s(0)) \\
&\to^6 s(s(0)) \times [s(0) \times fact(0)] \\
&\to^3 s(s(0)) \times [s(0) \times s(0)] \\
&\to^5 s(s(0)) \times [s(0) + (0 \times s(0))] \\
&\to^2 s(s(0)) \times [s(0) + 0] \\
&\to^4 s(s(0)) \times [s(0 + 0)] \\
&\to^1 s(s(0)) \times s(0) \\
&\to^5 s(0) + [s(0) \times s(0)] \\
&\to^5 s(0) + [s(0) + (0 \times s(0))] \\
&\to^2 s(0) + [s(0) + 0] \\
&\to^4 s(0) + s(0 + 0) \\
&\to^1 s(0) + s(0) \\
&\to^4 s(0 + s(0)) \\
&\to^1 s(s(0)).
\end{aligned}$$

Assim $fact(s(s(0)))$ reduz para $s(s(0))$. Observe que nenhuma das regras $(1) - (6)$ do Sistema de Reescrita se aplica ao termo $s(s(0))$. Neste caso, quando nenhuma das regras do Sistema de Reescrita se aplica a um dado termo, então dizemos que este termo está em sua *forma normal*. Aproveitamos ainda o momento, para introduzir um dos conceitos fundamentais da Teoria de Reescrita de Termos, que é: Um Sistema de Reescrita de Termos é *terminante* quando não se pode aplicar as regras de reescrita indefinidamente, ou seja, para qualquer que seja a cadeia (sequência) de aplicações das regras, sempre depois de um número finito de aplicações, chegamos a um termo para qual nenhuma das regras se aplicam mais (forma normal).

Diferenciação: Seja a inteiro, e f e g símbolos de função. Então um Sistema de Reescrita para diferenciação pode ser dado como:

$$\begin{array}{lrcl} (\text{D1}) & D(a) & \rightarrow & 0 \\ (\text{D2}) & D(x) & \rightarrow & 1 \\ (\text{D3}) & D(f+g) & \rightarrow & D(f)+D(g) \\ (\text{D4}) & D(f\cdot g) & \rightarrow & g\cdot D(f)+f\cdot D(g) \end{array}$$

Como exemplo de aplicação vamos calcular a derivada de $x \cdot x$ tal como apresentado na Figura 1.1.1.

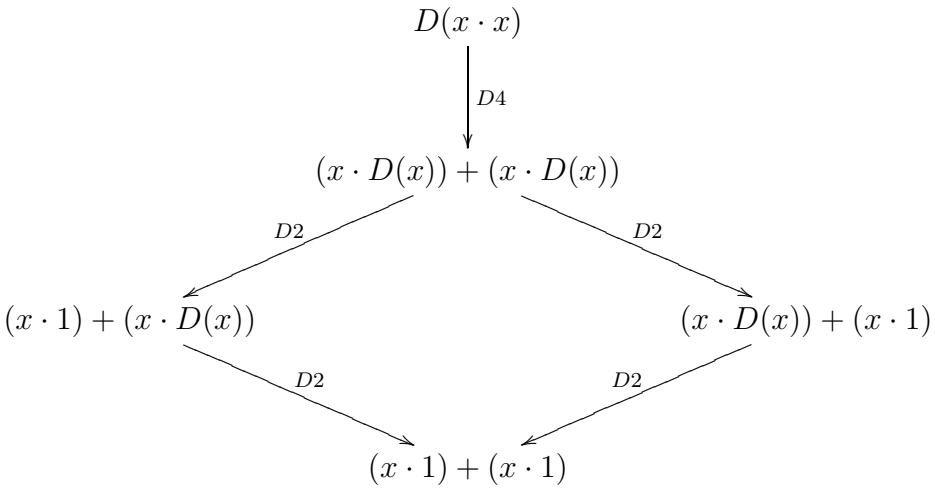


Figura 1.1.1: Cálculo da derivada de $x \cdot x$.

Com base neste exemplo podemos apresentar um outro conceito, também, considerado essêncial na Teoria de Reescrita de Termos, a saber: *confluência*. Observando a Figura 1.1.1 vemos que a partir do termo $(x \cdot D(x)) + (x \cdot D(x))$ podemos seguir dois caminhos diferentes, mas chegando ao mesmo termo $(x \cdot 1) + (x \cdot 1)$. Quando isto acontece, ou seja, quando diferentes caminhos de aplicações das regras a um dado termo t nos leva a dois diferentes termos t_1 e t_2 a partir dos quais podemos, através de aplicações das regras, sempre encontrar um termo comum t_0, dizemos que o sistema é confluente. É possível provar que o sistema de regras de reescrita $(D1) - (D4)$ é confluente [4].

Teoria dos Grupos: Seja $\circ$ um símbolo de função binária, $^{-1}$ um símbolo de função unária, e um símbolo de constante, e x, y e z símbolos de variáveis. Abaixo consideramos um Sistema de Reescrita que representa os seguintes axiomas (padrão) da Teoria de Grupos: $e \circ x = x$, $x^{-1} \circ x = e$ e $(x \circ y) \circ z = x \circ (y \circ z)$.

$$
\begin{array}{lrcl}
(\mathrm{G1}) & e \circ x & \to & x \\
(\mathrm{G2}) & x^{-1} \circ x & \to & e \\
(\mathrm{G3}) & (x \circ y) \circ z & \to & x \circ (y \circ z)
\end{array}
$$

Usando as regras de reescrita $(G1) - (G3)$ não podemos verificar que $y^{-1} \circ (e \circ x)^{-1} = ((e \circ x) \circ y)^{-1}$, como mostra a Figura 1.1.2, mesmo sabendo que esta igualdade é verdadeira na Teoria dos Grupos. Note que a verificação de uma igualdade entre termos utilizando um Sistema de Reescrita de Termos está intimamente ligada com a noção de simplificação (redução) de termos.

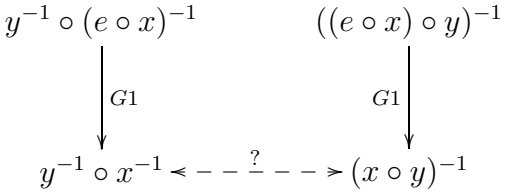


Figura 1.1.2: Exemplo de não confluência.

Observe que na Figura 1.1.2 nenhuma das regras $(G1) - (G3)$ se aplicam a $y^{-1} \circ x^{-1}$ e $(x \circ y)^{-1}$ e por isso não podemos *juntá-los*, ou seja, verificar a igualdade desejada. Abusando um pouco da linguagem, podemos dizer que estes dois termos formam um *par crítico* não juntável. E é ai que entra um método prático, chamado de *completação de Knuth-Bendix* [43], para converter pares críticos não juntáveis em regras de reescrita. A completação usa uma ordem parcial, chamada de *ordem de redução* [4], para orientar as equações, reescrever para simplificar regras e equações, e ordenar a fim de determinar qual das duas regras é a mais geral, e, por isso, a preferida. Knuth e Bendix [43] utilizaram seu método de completação e escreveram em Fortran um programa para completar Sistemas de Grupos Livres, Grupoídes dentre outros. Abaixo apresentamos um Sistema de Reescrita de Termos para a Teoria dos Grupos gerado pelo método de completação de Knuth e Bendix, e que nos permite provar a igualdade $y^{-1} \circ (e \circ x)^{-1} = ((e \circ x) \circ y)^{-1}$ (ver Figura 1.1.3).

$$
\begin{array}{lrcl@{\qquad}lrcl}
(\mathrm{G1}) & e \circ x & \to & x & (\mathrm{G6}) & x \circ e & \to & x \\
(\mathrm{G2}) & x^{-1} \circ x & \to & e & (\mathrm{G7}) & x \circ x^{-1} & \to & e \\
(\mathrm{G3}) & (x \circ y) \circ z & \to & x \circ (y \circ z) & (\mathrm{G8}) & e^{-1} & \to & e \\
(\mathrm{G4}) & (x \circ y)^{-1} & \to & y^{-1} \circ x^{-1} & (\mathrm{G9}) & (x^{-1})^{-1} & \to & x \\
(\mathrm{G5}) & x \circ (x^{-1} \circ y) & \to & y & (\mathrm{G10}) & x^{-1} \circ (x \circ y) & \to & y
\end{array}
$$

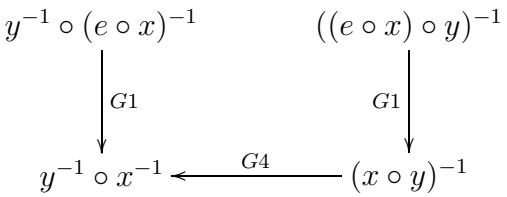


Figura 1.1.3: Par crítico juntável.

1.2 Contribuição

Em geral, técnicas da Teoria de Reescrita de Termos têm se mostrado adequadas e importantes em vários contextos matemáticos e computacionais. Em particular, operações básicas da reescrita, como casamento, redução, e substituição, são usadas nas linguagens de programação em geral, e para desenvolver procedimentos de prova em assistentes (semi)automáticos de prova, como por exemplo, ACL2 [39], Coq [5] e PVS [51]. Por outro lado, vêm-se usando os assistentes de prova para *formalizar* os conceitos e resultados da Teoria de Reescrita, e neste trabalho apresenta-se uma contribuição neste sentido: formalizam-se *teorias*, chamadas `ars` e `trs`, no assistente de prova PVS, onde são especificados, respectivamente conceitos, definições e resultados inerentes às teorias de ARS e de TRS.

Observamos que o intuito deste trabalho não é mostrar a eficiência e abrangência de aplicação do PVS, muito menos formalizar resultados específicos das teorias ARS e TRS, mas sim fornecer um conjunto de especificações que sirva de base para a formalização de propriedades ou resultados da Teoria de Reescrita. No entanto, para evidenciar que as teorias desenvolvidas, `ars` e `trs`, realmente desempenham o papel desejado, formalizamos diversos teoremas bem-conhecidos e não triviais tais como o Lema de Newman e o Teorema dos Pares Críticos de Knuth-Bendix. A formalização deste último, apresentada neste trabalho, até onde sabemos, é a primeira formalização completa deste teorema numa linguagem de especificação de ordem superior como o PVS.

1.3 Trabalhos Relacionados

Há diversos outros trabalhos sobre formalização e aplicação de conceitos relacionados com Teoria de Reescria já desenvolvidos. Por exemplo, em [35] Huet especifica e formaliza em Coq propriedades envolvendo confluência para o Cálculo Lambda, em particular para β redução. O principal resultado é uma formalização do Teorema do Prisma (Ver [70], Theorem 5, Prism). Em [59], Rasmussen apresenta uma translação para Isabelle do tratamento desenvolvido por Huet em Coq. Em [48], Nipkow trata de conceitos como confluência e comutação, e formaliza em Isabelle/HOL [49] alguns resultados, por exemplo, o Teorema da União Comutativa e Teoremas de Church-Rosser para as reduções β, η e $\beta \cup \eta$ do Cálculo Lambda não-tipado. Em [64], Shankar usando o Boyer-Moore [9], formaliza o Teorema de Church-Rosser para o Cálculo Lambda. A formalização do Cálculo Lambda usa índices de *de Bruijn* e a prova do teorema é baseada na versão de Tait-Martin-Löf, ou seja, é baseada na noção de redução paralela. Em [57], Pfenning apresenta uma formalização do Cálculo Lambda não-tipado, e formaliza a Propriedade de Church-Rosser em LCF [27]. Também, uma formalização em PVS do Teorema de Church-Rosser para uma versão do Cálculo Lambda *call-by-value* é apresentada por Ford e Mason em [18].

Em [46] McKinna e Pollack apresentam um *survey* sobre conceitos e resultados do Cálculo Lambda e Sistemas de Tipos Puros formalizados em LEGO [45]. Também, em [1] foi formalizado em LEGO, por Altenkirch, o Sistema F de Girard com principal objetivo de verificar que tal sistema é fortemente normalizável. Outros *Cálculos* formalizados, em Coq, com principal objetivo de verificar que são fortemente normalizáveis, são: Cálculo de Construções [10], [2]; Cálculo Lambda tipado com co-produtos [3]; e Cálculo Lambda Simples Tipado à la Church com constantes [44].

As bibliotecas *CoLoR* [7] e *Coccinelle* [13] desenvolvidas em Coq, respectivamente, por Blanqui et al e Contejean et al, limitam-se a formalizar critérios para terminação de TRS por meio de ordens de redução.

Em [63], Saïbi apresenta especificações em Coq de conceitos da teoria de reescrita, por exemplo, fecho de relações e confluência local, e formalizações de alguns resultados, por

exemplo, o Lema de Newman e o Lema de Yokouchi. Por outro lado, sem formalizar o teorema dos pares críticos de Knuth-Bendix, em [63] analisam-se os pares críticos do Cálculo de Substituições Explicitas $\lambda_{\sigma\Uparrow}$ e aplica-se axiomaticamente o teorema para verificar que esse cálculo é localmente confluente.

Diferentemente dos trabalhos anteriores, as *teorias* `ars` e `trs` pretendem ser mais gerais no sentido de incluir os elementos necessários para formalização de resultados da Teoria da Reescrita, sem se fixar num sistema de reescrita ou cálculo em particular.

Em [61], Ruiz-Reina et al apresentam uma formalização em ACL2, usando lógica de primeira-ordem, de conceitos e resultados sobre ARS e TRS. Sendo assim, dentre os trabalhos citados acima e que conhecemos, o trabalho [61] é o que mais se aproxima do desenvolvimento apresentado aqui no sentido de ser uma formalização geral da Teoria de Reescrita de Termos. Porém, há diferenças entre nossa abordagem e este trabalho, por exemplo, entre as lógicas usadas: a formalização em ACL2 usa a lógica de primeira-ordem, ao contrário de `ars` e `trs` que foram desenvolvidas, aproveitando a linguagem de ordem superior do sistema PVS, em uma lógica de segunda-ordem. Dessa forma, nossa formalização utiliza quantificações de objetos relacionais como as próprias relações de reescrita, o que possibilita, além de elegância, um nível de abstração que nos permite expressar propriedades das relações de forma quase geométrica, como desejável em Teoria de Reescrita, em particular, em ARS.

Uma das características do desenvolvimento apresentado neste trabalho é o uso de *variáveis com nomes* em vez de *índices de de Bruijn*. Alguns dos trabalhos citados acima, por exemplo [64] e [35], usam índices de de Bruijn em suas formalizações para evitar formalizar *renomeamento de variáveis* o qual previne capturas indesejáveis durante a aplicação de substituições. Ao contrário de [46], [18] e [61] que apresentam suas formalizações baseadas em variáveis com nomes.

Apesar da notação de de Bruijn ser muito elegante e conveniente para a formalização do Cálculo Lambda, o seu uso se torna incoveniente para representar sistemas de reescrita em geral, ao contrário da formalização usando variáveis com nomes que permite representar elementos matemáticos de uma forma natural, por exemplo, o uso de funções, isto é, são

apresentadas de uma forma muito próxima daquelas apresentadas nos livros texto.

1.4 Organização

No decorrer de todo este texto quando usamos a palavra *teoria* em itálico estamos nos referindo à teoria desenvolvida no assistente de prova PVS, e quando usamos a palavra *formalização* estamos nos referindo à *prova mecânica* desenvolvida no assistente de prova PVS.

Nos Capítulos 2 e 3 apresentamos, respectivamente, visões gerais sobre a Teoria da Prova e a semântica do PVS. Já nos Capítulos 4 e 5 apresentamos as *teorias* `ars` e `trs`, respectivamente. Nestes capítulos, detalhes das provas em PVS, algumas definições e teoremas são omitidos; porém, os desenvolvimentos completos das *teorias* `ars` e `trs` estão disponíveis em `www.mat.unb.br/~ayala/publications.html`. Em seguida, apresentamos a conclusão e trabalhos futuros. Nos Apêndices A, B e C, apresentamos alguns detalhes sobre a linguagem de especificação, sobre o assistente de prova, e sobre comandos (estratégias) de prova do PVS, respectivamente.

Capítulo 2

Elementos Básicos da Teoria da Prova

Em meados do século XIX e início do século XX, com o desenvolvimento dos Métodos Algébricos e da Teoria dos Conjuntos assim como com o descobrimento de diversos paradoxos envolvendo conceitos conjuntistas, a fundamentação da matemática passou a ser o foco central de três escolas denominadas: *Logicismo*, *Intuicionismo* e *Formalismo*. A escola de Formalismo, liderada por David Hilbert, pretendia provar a consistência de axiomatizações da matemática através de métodos considerados recursivos, finitistas e computacionais [41]. Originalmente concebido por Hilbert em torno do conceito de decidibilidade, o estudo das propriedades estruturais de provas formais constitui o cerne da pesquisa relacionada à Teoria da Prova Estrutural, ou simplesmente Teoria da Prova [68], a qual pode ser descrita como o estudo da estrutura geral das provas matemáticas, e de argumentos com força demonstrativa como encontrada em lógica. A caracterização da noção de prova normal, os teoremas de existência, unicidade e consistência de tais provas, o estabelecimento de limites para o tamanho da prova de uma dada fórmula são, dentre outros, resultados alcançados nesta área. O desenvolvimento de provadores (semi)automáticos de teoremas e da programação em lógica e funcional tem sido impulsionado por tais resultados [24].

Em 1958, H. B. Curry observou que existia uma correspondência sintática entre o sistema de dedução de Hilbert e a lógica combinatorial [16]. A partir daí, em 1969 [33], W. A. Howard mostrou explicitamente uma correspondência sintática entre os programas do *cálculo lambda tipado simples* [29] e as provas da *dedução natural*. Esta correspondência é conhecida na literatura como o *Isomorfismo de Curry-Howard* [67], em outras palavras, este isomorfismo estabelece, de maneira geral, uma correspondência entre a noção de

prova e a noção de computação. Vale ressaltar que observação similar pode ser feita com relação ao *cálculo de sequentes*. Dessa forma, a *Teoria dos Tipos* [24] vem de encontro com o propósito geral da Teoria da prova que é o estudo das estruturas, a comparação, a identificação e a distinção das provas.

2.1 Sistemas de Gentzen

O *Programa de Hilbert* propõe que a consistência de sistemas mais complexos, como análise real, poderiam ser provados em termos de sistemas mais simples. Assim, a consistência de toda a matemática seria reduzida à aritmética básica. No entanto, o *Teorema da Incompletude*[1] de Gödel [17] vem devastar esta idéia mostrando que a aritmética básica não pode ser usada para provar sua própria consistência, e consequentemente não pode ser usada para provar a consistência de nada mais forte. Neste momento, sem dúvida nenhuma, reina uma insatisfação com relação aos sistemas de demonstração formal existentes, criados não só por Hilbert, mas também por Frege e Russell. Talvez essa insatisfação levou à publicação, em 1934/5, de dois artigos que até hoje servem como base para o ensino de lógica elementar. Estes dois trabalhos foram publicados por dois autores que, aparentemente, nunca tinham se encontrado antes, e discursavam sobre um novo procedimento em lógica chamado *dedução natural*. Estes autores eram *S. Jaskowski* [37] e *G. Gentzen* [23].

Juntamente com a dedução natural, Gentzen propôs um sistema dedutivo extremamente rico usado não somente para construir provas mas támbem para estudar as metapropriedades dos sistemas lógicos sob a perspectiva da Teoria da Prova, a saber: O *Cálculo de Sequentes* [23]. Devido a certas assimetrias inerentes às regras da dedução natural, Gentzen propôs o cálculo de sequentes com o intuito principal de demonstrar o *Hauptsatz*, também chamado de *Teorema da Eliminação do Corte*, para a lógica clássica de forma mais conveniente. Este teorema nos diz que toda prova pode ser transformada em uma prova normal, canônica, padrão.

[1]Em 1931, o matemático Kurt Gödel provou o chamado *Teorema da Incompletude* sobre a natureza da matemática. O teorema afirma que, dentro de qualquer eixo formal de axiomas, como a matemática atual, sempre persistem questões que não podem ser provadas e nem refutadas

Os sistemas de cálculo de sequentes e dedução natural possuem algumas vantagens em relação à apresentação axiomática tradicional. No sistema axiomático, existem muitas possibilidades de escolha de axiomas para definir um dado símbolo lógico e, além disso, a definição de um símbolo lógico é frequentemente apresentada por um axioma onde outros símbolos lógicos aparecem. O sistema de Hilbert [41], por exemplo, apresenta axiomas para definir o papel dedutivo de cada símbolo lógico no qual a implicação é sempre utilizada. Gentzen foi capaz de apresentar o papel dedutivo de uma forma individualizada descartando o papel da implicação dos outros símbolos lógicos. Neste sentido, tanto no sistema de dedução natural quanto no de cálculo de sequentes, as inferências são quebradas em passos atômicos, ou seja, cada regra de inferência trata de apenas um símbolo lógico.

2.1.1 Dedução Natural

Uma maneira intuitiva de apresentar as demonstrações em dedução natural é através de *árvores de derivação* (*árvores de dedução*). De outro modo, o sistema de dedução natural apresenta regras que unem árvores (finitas) que são geradas a partir de um conjunto finito de premissas e hipóteses até derivar uma certa conclusão, ou seja, cada passo (ou cada derivação) realizada na árvore deve ser baseada em uma das regras do sistema, e é aí que reside a principal característica do sistema de Dedução Natural, ou seja, para cada conectivo lógico ($\vee, \wedge, \rightarrow, \bot, \top$) existem regras de *introdução* (exeto para $\bot$) e regras de *eliminação* (exeto para $\top$) [67] as quais são apresentadas na Figura 2.1.1.

Definição 2.1.1:

1. Um *juízo* em dedução natural é um par, denotado por $\Gamma \vdash \varphi$ (lê se: Γ prova φ ou de Γ deduz-se φ), consistindo de um conjunto finito de fórmulas Γ e uma fórmula φ. Para fim de simplificação escrevemos, respectivamente:

$$\begin{array}{r} \psi_1, \psi_2 \vdash \varphi \\ \Gamma, \Delta \vdash \varphi \\ \vdash \varphi \end{array} \qquad \text{em vez de} \qquad \begin{array}{r} \{\psi_1, \psi_2\} \vdash \varphi \\ \Gamma \cup \Delta \vdash \varphi \\ \emptyset \vdash \varphi \end{array}$$

2. Uma *prova formal* de $\Gamma \vdash \varphi$ é uma árvore finita de juízos satisfazendo as seguintes condições:

Axioma:

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ (Ax)$$

Verdadeiro ($\top$) e Falso ($\bot$):

$$\frac{}{\Gamma \vdash \top}\ (\top I) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ (\bot E)$$

Negação:

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg I) \qquad \frac{\Gamma \vdash \neg\varphi \quad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg E)$$

Conjunção:

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ (\wedge I) \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ (\wedge E) \quad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\ (\wedge E)$$

Disjunção:

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ (\vee I) \quad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ (\vee I) \qquad \frac{\Gamma, \varphi \vdash \theta \quad \Gamma, \psi \vdash \theta \quad \Gamma \vdash \varphi \vee \psi}{\Gamma \vdash \theta}\ (\vee E)$$

Implicação:

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to I) \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to E)$$

Figura 2.1.1: Regras da Dedução Natural (Clássico)

(a) A raiz da árvore é a conclusão, ou seja, $\Gamma \vdash \varphi$;

(b) Todas as folhas da árvore são axiomas, ou seja, juízos da forma $\Gamma, \varphi \vdash \varphi$;

(c) Todo nó pai é obtido de nós filhos usando uma das regras apresentadas na Figura 2.1.1. Na verdade, os filhos são as derivações que geram a conclusão.

Se tal prova existe, dizemos que o juízo $\Gamma \vdash \varphi$ é *provável* ou *derivável*.

O sistema de prova consiste de um esquema de axiomas e regras. Para cada conectivo lógico (exceto para $\bot$ e $\top$) temos uma ou duas regras de introdução, e uma ou duas regras de eliminação. Uma regra de introdução para um conectivo $\Diamond$ nos diz como uma conclusão da forma $\varphi \Diamond \psi$ pode ser derivada, em outras palavras, as regras de introdução introduzem conectivos lógicos nas derivações, e elas são melhor utilizadas quando estamos construindo uma derivação a partir da conclusão e em direção às hipóteses ("de baixo para cima"). Agora uma regra de eliminação descreve o caminho para o qual $\varphi \Diamond \psi$ pode ser usada para derivar outras fórmulas; de outra forma, as regras de eliminação mostram como retirar os conectivos para podermos gerar derivações, e elas são melhor utilizadas quando estamos construindo uma derivação a partir das hipóteses e em direção à conclusão ("de cima para baixo").

Em particular, a premissa $\Gamma, \varphi \vdash \psi$ da regra $(\to I)$ pode ser entendida como a habilidade de inferir ψ de Γ se uma prova de φ é fornecida. Isto é o suficiente para derivar a implicação $(\to)$. A regra de eliminação $(\to E)$ possui a mesma idéia: se temos uma prova de $\varphi \to \psi$ então podemos transformar uma prova de φ em uma prova de ψ. Neste sentido, a regra $(\to E)$ pode ser vista como a inversa da regra $(\to I)$. Uma obsevação similar pode ser lançada sobre os outros conectivos. A regra $\bot E$, chamada de *ex falso*, é uma exceção, pois não existe uma regra de introdução correspondente, assim como, não existe uma regra de eliminação correspondente à regra $\top I$.

Exemplo 2.1.1: Vejamos alguns exemplos de prova (dedução). Abaixo as fórmulas φ, ψ e θ podem ser arbitrárias.

(i)

$$\frac{\varphi \vdash \varphi}{\vdash \varphi \to \varphi}\ (\to I)$$

(ii)

$$\dfrac{\dfrac{\varphi, \psi \vdash \varphi}{\varphi \vdash \psi \to \varphi}\,(\to I)}{\vdash \varphi \to (\psi \to \varphi)}\,(\to I)$$

(iii) Neste exemplo, usamos Γ para abreviar o conjunto $\{\varphi \to (\psi \to \theta), \varphi \to \psi, \varphi\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \varphi \to (\psi \to \theta) \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi \to \theta}\,(\to E) \quad \dfrac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\,(\to E)}{\Gamma \vdash \theta}\,(\to E)}{\varphi \to (\psi \to \theta), \varphi \to \psi \vdash \varphi \to \theta}\,(\to I)}{\varphi \to (\psi \to \theta) \vdash (\varphi \to \psi) \to (\varphi \to \theta)}\,(\to I)}{\vdash (\varphi \to (\psi \to \theta)) \to (\varphi \to \psi) \to (\varphi \to \theta)}\,(\to I)$$

2.1.2 Cálculo de Sequentes

O Cáculo de Sequentes, que iremos apresentar abaixo, difere um pouco da Dedução Natural, e estes dois sistemas servem para diferentes propósitos. De fato, na Dedução Natural vemos em destaque as propriedades fundamentais dos conectivos através de suas regras de introdução e de eliminação para cada conectivo. Já o cálculo de sequentes é, por assim dizer, mais prático; em outras palavras, mais eficiente com relação à construção de prova. As regras do cálculo de sequentes, as quais são lidas de baixo para cima, definem métodos para repassar uma fórmula complexa por uma outra mais simples; ou seja, reduz o questionamento inicial à verificação de axiomas. Em vez de regras de introdução e eliminação, existem somente regras de introdução. Estas regras de introdução podem introduzir conectivos tanto na parte de conclusão quanto na parte de suposições de um juízo, estas últimas substituem as regras de eliminação da Dedução Natural. Quando as regras introduzem conectivos na conclusão, estas se comportam exatamente igual às regras de introdução da Dedução Natural.

O cálculo de sequentes alcançou um certo destaque não só nas aplicações teóricas como também nas aplicações práticas, como por exemplo, no desenvolvimento de provadores (semi)automáticos de teoremas para os quais o cálculo de sequentes serve de base, tal como o assistente de prova *Prototype Verification System* (PVS) [51].

Existem muitas variações com respeito à definição de Cálculo de Sequentes, no entanto,

Axioma e Regra do Corte:

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ (Ax) \qquad \frac{\Gamma \vdash \varphi, \Delta \quad \Gamma, \varphi \vdash \Sigma}{\Gamma \vdash \Delta, \Sigma}\ (Corte)$$

Regras para $\bot$ e $\top$:

$$\frac{}{\Gamma, \bot \vdash \Delta}\ (L\bot) \qquad \frac{}{\Gamma \vdash \top, \Delta}\ (R\top)$$

Regras Estruturais:

$$\frac{\Gamma \vdash \Delta}{\Gamma, \varphi \vdash \Delta}\ (LW) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \varphi, \Delta}\ (RW)$$

$$\frac{\Gamma, \varphi, \psi, \Gamma' \vdash \Delta}{\Gamma, \psi, \varphi, \Gamma' \vdash \Delta}\ (LX) \qquad \frac{\Gamma \vdash \Delta, \varphi, \psi, \Delta'}{\Gamma \vdash \Delta, \psi, \varphi, \Delta'}\ (RX)$$

$$\frac{\Gamma, \varphi, \varphi \vdash \Delta}{\Gamma, \varphi \vdash \Delta}\ (LC) \qquad \frac{\Gamma \vdash \varphi, \varphi, \Delta}{\Gamma \vdash \varphi, \Delta}\ (RC)$$

Regras Lógicas:

$$\frac{\Gamma, \varphi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta}\ (L\wedge) \quad \frac{\Gamma, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta}\ (L\wedge) \qquad \frac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta}\ (R\wedge)$$

$$\frac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta}\ (L\vee) \qquad \frac{\Gamma \vdash \varphi, \Delta}{\Gamma \vdash \varphi \vee \psi, \Delta}\ (R\vee) \quad \frac{\Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \vee \psi, \Delta}\ (R\vee)$$

$$\frac{\Gamma \vdash \varphi, \Delta \quad \Gamma, \psi \vdash \Sigma}{\Gamma, \varphi \to \psi \vdash \Delta, \Sigma}\ (L\to) \qquad \frac{\Gamma, \varphi \vdash \psi, \Delta}{\Gamma \vdash \varphi \to \psi, \Delta}\ (R\to)$$

Figura 2.1.2: Regras do Cálculo de Sequentes (Clássico)

usaremos aqui aquela tal como estudada em [67].

Definição 2.1.2:

1. Um *sequente* é um par (Γ, Δ), denotado por $\Gamma \vdash \Delta$, onde Γ e Δ são sequências finitas de fórmulas.

2. Uma *derivação* de $\Gamma \vdash \Delta$ é uma árvore de sequentes, onde

   (a) a raiz é $\Gamma \vdash \Delta$;

   (b) os pais e filhos casam, respectivamente, com os sequentes abaixo e acima da linha das regras apresentadas na Figura 2.1.2;

   (c) todas as folhas da árvore devem ser axiomas.

O significado intuitivo de $\Gamma \vdash \Delta$ é o seguinte: a conjunção de todas as fórmulas em Γ implica a disjunção de todas as fórmulas em Δ, ou seja, se $\Gamma = \varphi_1, \varphi_2, \cdots, \varphi_n$ e $\Delta = \psi_1, \psi_2, \cdots, \psi_m$ então,

$$\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n \quad \Rightarrow \quad \psi_1 \vee \psi_2 \vee \cdots \vee \psi_m$$

Na Figura 2.1.2 (LW), (RW), (LX), (RX), (LC) e (RC) significam, respectivamente, as regras de enfraquecimento, permutação e contração à esquerda (L) e à direita (R). Se considerarmos o lado direito da regra $(R \rightarrow)$ como sendo um conjunto unitário e o lado esquerdo como sendo um conjunto arbitário é fácil ver que esta regra é idêntica à regra de introdução na Dedução Natural. Observação análoga pode ser lançada sobre os outros conectivos. No entanto, note que a regra $(L \rightarrow)$ não pode ser comparada com a regra de eliminação na Dedução Natural. Vejamos agora alguns exemplos:

Exemplo 2.1.2: 1. A seguir apresentamos uma derivação para

$$\vdash (p \rightarrow q) \rightarrow (q \rightarrow r) \rightarrow (p \rightarrow r)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{p \vdash p \quad q \vdash q}{p, p \to q \vdash q}\,(L\to)}{p \to q, p \vdash q}\,(LX) \qquad \dfrac{\dfrac{\dfrac{r \vdash r}{r, p \to q \vdash r}\,(LW)}{p \to q, r \vdash r}\,(LX)}{p \to q, r, p \vdash r}\,(LW)}{p \to q, p, r \vdash r}\,(LX)}{p \to q, p, q \to r \vdash r}\,(L\to)$$
$$\dfrac{p \to q, q \to r, p \vdash r}{p \to q, q \to r \vdash p \to r}\,(LX)$$
$$\dfrac{(p \to q) \vdash (q \to r) \to (p \to r)}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\,(R\to)$$

2. Damos a seguir um prova para a Lei de Pierce:

$$\dfrac{\dfrac{\dfrac{\dfrac{p \vdash p}{p \vdash q, p}\,(RW)}{\vdash p \to q, p}\,(R\to) \qquad p \vdash p}{(p \to q) \to p \vdash p, p}\,(L\to)}{(p \to q) \to p \vdash p}\,(RC)$$
$$\dfrac{}{\vdash ((p \to q) \to p) \to p}\,(R\to)$$

2.2 Sistemas de Tipos

Inventada por Bertrand Russel [62] em 1908, a Teoria dos Tipos é uma linguagem para fomalização matemática, e veio, em parte, para responder ao *paradoxo de Russell*: Seja U o conjunto de todos os conjuntos que não contém a si próprios como membros, ou seja, $U = \{A \mid A \notin A\}$. O que podemos dizer sobre U, isto é, U contém a si mesmo? Se sim, pela própria definição de U, U não é membro de U. Se não (U não é membro de si mesmo), então obrigatoriamente U deve ser membro de U, por definição de U. Dessa forma, qualquer que seja a resposta, *sim* ou *não*, obtemos uma contradição.

Assim, Russel esquivou-se deste paradoxo criando uma hierarquia infinita, chamada de *Teoria dos Tipos*, de tal modo que um conjunto não pode ser membro de si mesmo, nem de qualquer conjunto de tipo inferior. Um *tipo* (σ) é simplesmente interpretado como um conjunto, e *tipos funcionais* ($\sigma \to \tau$) são interpretadas como um conjunto de funções entre conjuntos (de σ para τ), ou seja, como uma relação funcional, um conjunto de pares de elementos. No entanto, não se deve pensar que tipos são conjuntos, pois os tipos nos dão uma informação *sintática* enquanto que conjuntos nos dão uma informação *semântica*.

Em 1940, Alonzo Church [11] apresentou uma nova formulação da Teoria dos Tipos, conhecida como *Teoria de Tipos à la Church*, que de certa forma é mais simples e mais geral do que aquela apresentada por Russel. Esta nova formulação é baseada sobre funções em vez de relações e possui um importante mecanismo para construir e aplicar funções. Vale ressaltar, que nas últimas décadas a Teoria dos Tipos tem sido uma escolha de consenso geral para o desenvolvimento de provadores (semi)automáticos de teoremas, por exemplo, HOL [36], Isabelle [49] e PVS [51] são sistemas baseados na Teoria de Tipos à la Church.

2.2.1 Cálculo Lambda : Visão Geral

Em 1930s, Church inventou o *Cálculo Lambda* (λ-cálculo) como parte de um sistema para lógicas de ordem superior e teoria das funções, com o intuito de capturar os aspectos mais básicos da maneira pela qual funções podem ser combinadas para formar outras funções. Em outras palavras, o λ-cálculo modela e estuda o comportamento aplicativo das funções e por isso tem como operação básica a *aplicação*.

Uma função no λ-cálculo é representada na forma $\lambda x.M$, onde x é o parâmetro da função e M constitui o corpo da função. No λ-cálculo a expressão $\lambda x.M$ pertence a um conjunto de elementos chamados de λ-*termos* e que definimos a seguir:

Definição 2.2.1: Seja Υ um conjunto enumerável infinito de símbolos, os quais chamamos de λ-*variáveis*, ou simplesmente, *variáveis*. Um λ-termo é um dentre os seguintes:

(i) cada *variável* é um λ-termo, chamado de **termo atômico**;

(ii) se M e N são λ-termos, então (MN) é um λ-termo, chamado de uma **aplicação**;

(iii) se M é λ-termo e x é uma variável, então $\lambda x.M$ é um λ-termo, chamado de uma **abstração** ou uma **λ-abstração**.

Em uma abstração $\lambda x.M$, a variável x, que representa o argumento da função, pode ou não ocorrer em M. No caso positivo, qualquer ocorrência de x em M é dita ser *ligada*, enquanto que qualquer outra variável é dita ser *livre*, a menos que esta outra variável seja

ligada por uma outra abstração. Por exemplo, a variável y em $\lambda x \lambda y.xy$ é ligada, enquanto que em $\lambda x.xy$ a mesma variável é livre. Se um λ-termo não possui variáveis livres, então dizemos que este é *fechado*. Já em uma aplicação (MN) não há qualquer restrição sobre o operador M e o argumento N.

Denotamos os λ-termos por letras maiúsculas do nosso alfabeto, por exemplo, L, M, N, P, Q, ... com ou sem índices, e nos referimos aos λ-termos simplesmente como *termos*. Também, por conveniência, repetidos $\lambda's$ e parênteses serão omitidos, e adotamos a *associação à esquerda* para recuperar os parênteses. Por exemplo,

$$\lambda xyz.M = (\lambda x.(\lambda y.(\lambda z.M))) \qquad MNPQ = (((MN)P)Q)$$

Dentre as regras de simplificação do λ-cálculo apresentamos as seguintes:

α-conversão: permite renomear variáveis ligadas. Por exemplo, $\lambda x.x$ pode ser α-convertido para $\lambda y.y$.

$$\lambda x.x \rightarrow_\alpha \lambda y.y$$

Note que a α-conversão nos permite considerar idênticos os λ-termos que só direfem nos nomes das variáveis ligadas, ou seja, podemos substituir uma variável x em M por qualquer outro termo N mudando, se necessário, os nomes de algumas variáveis ligadas em M.

β-redução: permite que aplicações sejam reduzidas. Por exemplo, $(\lambda x.M)N$ pode ser β-reduzido para $M[x/N]$ no qual todas as ocorrências da variável x foram substituidas por N. Porém, se houver conflito de nomes de variáveis, como em $(\lambda x.\lambda y.xy)y$, deve-se primeiro aplicar a α-conversão.

$$(\lambda x.M)N \rightarrow_\beta M[x/N]$$

Um termo M está em *β-forma normal*, ou simplesmente em *forma normal*, se, e somente se, não existe um termo N tal que $M \rightarrow_\beta N$.

Exemplos 2.2.1:

1. A função identidade é dada por:

$$\lambda x.x$$

De fato,

$$(\lambda x.x)M \rightarrow_\beta M$$

2. Expressamos os números naturais no λ-cálculo como os *numerais de Church*:

$$c_n = \lambda yx.y^n(x)$$

onde $y^n(x)$ é uma abreviação para $y(y(\cdots(yx)\cdots))$ com n ocorrências de y.

Em geral, para um número natural n, escrevemos $\mathbf{n}$ em vez de c_n. Por exemplo,

$$\mathbf{0} = \lambda yx.x \qquad \mathbf{1} = \lambda yx.yx \qquad \mathbf{2} = \lambda yx.y(yx)$$

3. Funções básicas:

Função Zero: $Z \equiv \lambda m.\mathbf{0}$

Função Sucessor: $S \equiv \lambda nyx.y(nyx)$

Função Soma: $A_+ \equiv \lambda mnyx.my(nyx)$

Função Multiplicação: $A_* \equiv \lambda mnyx.m(ny)x$

Função Exponencial: $A_{exp} \equiv \lambda mnyx.mnyx$

Função Projeção: $\Pi_i^k \equiv \lambda m_1 \cdots m_i \cdots m_k.m_i$

4. Uma função parcial $f : \mathbb{N}^m \rightarrow \mathbb{N}$ é λ-*definível* se, e somente se, existe um λ-termo F tal que as seguintes se verificam:

   (a) Se $f(n_1, n_2, \cdots, n_k) = m$ então $Fc_{n_1}c_{n_2}\cdots c_{n_k} =_\beta c_m$. Onde $=_\beta$ é a menor relação de equivalência contendo $\rightarrow_\beta$.

   (b) Se $f(n_1, n_2, \cdots, n_k) = m$ é indefinida então $Fc_{n_1}c_{n_2}\cdots c_{n_k}$ não possui forma normal.

5. Considere o termo $(\lambda x.xx)(\lambda x.xx)$. Observe que:

$$(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \cdots ,$$

ou seja, o $(\lambda x.xx)(\lambda x.xx)$ não possui uma forma normal, ou melhor, o λ-cálculo não é terminante. Por outro lado, podemos encarar a forma normal como o resultado final da computação, de outra forma, se uma redução termina, então o resultado é único, como mostra o seguinte resultado[2]: Como usual, a relação $\rightarrow_\beta^*$ representa o fecho reflexivo transitivo de $\rightarrow_\beta$.

Teorema 2.2.2 [Teorema Church-Rosser]: Se $M \rightarrow_\beta^* N_1$ e $M \rightarrow_\beta^* N_2$, então existe um termo L tal que $N_1 \rightarrow_\beta^* L$ e $N_2 \rightarrow_\beta^* L$.

Corolário 2.2.3: Se M possui uma forma normal, então essa forma normal é única.

Apesar do λ-cálculo não ter tido o mesmo impacto das *Máquinas de Turing*[3] [32] e das enormes diferenças de notação e enfoque, provou-se que há uma equivalência entre a abordagem de Turing e a de Church, ou seja, entre as máquinas de Turing e o λ-cálculo. Esta equivalência dos formalismos deu origem à chamada *Tese de Church-Turing*[4], que afirma: "tudo que é computável", é "computável" a partir desses dois mecanismos, e qualquer outro tipo de "computabilidade" que possa ser inventada ou será menos expressiva ou equivalente à computabilidade deles. Em outras palavras, λ-cálculo é um formalismo universal no sentido de que toda função computável pode ser expressa usando este formalismo. Essa expressividade do λ-cálculo é constatada através do seguinte teorema, cuja demonstração pode ser encontrada em [67].

Teorema 2.2.4: Todas as funções recursivas são λ-definíveis.

O sistema de λ-cálculo, apresentado acima, é hoje chamado de *cálculo lambda livre de tipos (sem tipos)* para distingui-lo de outras versões do λ-cálculo com tipos. Nas versões

[2]Provas desses resultados podem ser encontradas em [29].

[3]Muito tempo antes de existirem os modernos computadores digitais de hoje, Alan Turing (1936) inventou uma classe de máquinas, hoje chamadas de máquinas de Turing e através delas definiu a noção de função computacional. Uma máquina de Turing nada mais é de que um modelo abstrato de um computador, o qual restringe-se a apenas aos aspectos lógicos do seu funcionamento (memória, estados e transições).

[4]A Tese de Church-Turing não pode ser provada formalmente por que a noção de função computável (em um sentido intuitivo) não é definida formalmente.

tipadas do λ-cálculo *tipos* são designados para os λ-termos. Dessa forma, um λ-termo M pode ser aplicado a um λ-termo N de tipo σ se, e somente se, M possui tipo $\sigma \to \tau$ para algum tipo τ. Assim, podemos ver um tipo φ como uma proposição e o λ-termo M associado a este tipo como sendo uma prova para esta proposição, e denotamos esta idéia por $M : \varphi$.

Entre as várias versões tipadas do λ-cálculo destacamos apenas duas, que de certa forma é como se dividem basicamente todas as outras, a saber: Os sistemas tipados *à la Church* [11] e os sistemas tipados *à la Curry* [15]. A principal diferença entre estes dois sistemas é que no λ-cálculo tipado à la Church o tipo de cada variável é fixado, e no λ-cálculo tipado à la Curry não é. Por exemplo, no λ-cálculo tipado à la Church o termo $\lambda x : \sigma.x$ é a função identidade sobre o tipo σ, e ele possui tipo $\sigma \to \sigma$, mas não $\tau \to \tau$ se o tipo τ for diferente do tipo σ. Já no λ-cálculo tipado á la Curry, o termo $\lambda x.x$ representa uma função identidade geral com tipo $\rho \to \rho$ para "todo" tipo ρ. Porém, tanto a tipagem à la Church quanto a tipagem à la Curry dependem de um conjunto $\mathbb{T}$ de tipos, o qual definimos a seguir [29].

Definição 2.2.5: Suponhamos que seja dado um conjunto infinito enumerável de *variáveis-tipo*, distintos das λ-variáveis. O conjunto de *tipos* $\mathbb{T}$ é definido indutivamente como segue:

(i) toda variável-tipo σ é um tipo, ou seja, $\sigma \in \mathbb{T}$;

(ii) se $\sigma, \tau \in \mathbb{T}$ então $(\sigma \to \tau)$ é um tipo, ou seja, $(\sigma \to \tau) \in \mathbb{T}$ (**tipo funcional**).

Por simplicidade o excesso de parênteses é omitido dos tipos e usamos a *associação à direita* para recuperá-los. Por exemplo,

$$\sigma \to \tau \to \rho \equiv (\sigma \to (\tau \to \rho))$$

Também por conveniência usaremos uma síntaxe abstrata para formar $\mathbb{T}$:

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$$

onde $\mathbb{V}$ é o conjunto de variáveis-tipo.

2.2.2 λ-cálculo tipado à la Curry

A designação de tipos à la Curry depende de dois parâmetros, o conjunto de tipos $\mathbb{T}$ e um conjunto de regras de designação de tipos.

Definição 2.2.6:

1. Uma *designação de tipo* é qualquer expressão da forma $M : \sigma$ (lê-se: M possui tipo σ) onde M é um λ-termo, chamado *sujeito*, e σ é um tipo, chamado *predicado*.

2. Um *contexto* Γ é qualquer conjunto finito, possivelmente vazio, de designações de tipos, $\Gamma = \{x_1 : \sigma_1, x_2 : \sigma_2, \cdots, x_n : \sigma_n\}$, onde os sujeitos são variáveis, e que seja consistente, ou seja, que nenhuma variável seja sujeito de mais de uma designação de tipo.

3. Uma designação $M : \sigma$ é derivável do contexto Γ, denotado por $\Gamma \vdash M : \sigma$, se $\Gamma \vdash M : \sigma$ pode ser construído a partir das regras apresentadas a seguir:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ (Ax)$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma \to \tau}\ (\to I)$$

$$\frac{\Gamma \vdash M : (\sigma \to \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}\ (\to E)$$

4. Seja o contexto $\Gamma = \{x_1 : \sigma_1, \cdots, x_n : \sigma_n\}$. Definimos $||\Gamma||$, chamado na literatura de *imagem*, como:

$$||\Gamma|| = \{\sigma \in \mathbb{T} \mid (x : \sigma) \in \Gamma, \text{para algum } x\}$$

Exemplos 2.2.2: Sejam σ, τ, ρ tipos arbitrários. Então:

1. $\vdash (\lambda xy.x) : \sigma \to \tau \to \sigma$. De fato,

$$\frac{\dfrac{x : \sigma, y : \tau \vdash x : \sigma}{x : \sigma \vdash (\lambda y.x) : \tau \to \sigma}\ (\to I)}{\vdash (\lambda xy.x) : \sigma \to \tau \to \sigma}\ (\to I)$$

2. $\vdash (\lambda x.x) : \sigma \to \sigma$

$$\frac{x : \sigma \vdash x : \sigma}{\vdash (\lambda x.x) : \sigma \to \sigma} \ (\to I)$$

3.

$$\frac{\dfrac{x : \sigma, y : \sigma \vdash x : \sigma}{y : \sigma \vdash (\lambda x.x) : \sigma \to \sigma} \ (\to I) \qquad y : \sigma \vdash y : \sigma}{y : \sigma \vdash (\lambda x.x)y : \sigma} \ (\to E)$$

2.2.3 λ-cálculo tipado à la Church

Como já comentamos antes, a idéia de tipagem do λ-cálculo à la Church é diferente da tipagem à la Curry, pois, na tipagem à la Church uma completa informação sobre os tipos estão contidas nos termos, ou seja, os tipos das variáveis e termos são "fixos", enquanto que na tipagem à Curry os termos são λ-termos do λ-cálculo sem tipos.

Sendo $\mathbb{T}$ um conjunto de tipos, definimos os chamados *Pseudotermos*, denotado por $\Lambda_{\mathbb{T}}$, como segue:

$$\Lambda_{\mathbb{T}} = V \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \lambda V : \mathbb{T}.\Lambda_{\mathbb{T}}$$

onde V é o conjunto de λ-variáveis.

Definição 2.2.7:

1. Uma *designação de tipo* é qualquer expressão da forma $M : \sigma$ com $M \in \Lambda_{\mathbb{T}}$ e $\sigma \in \mathbb{T}$.

2. Um *contexto* Γ é qualquer conjunto finito, possivelmente vazio, de designações de tipos, $\Gamma = \{x_1 : \sigma_1, x_2 : \sigma_2, \cdots, x_n : \sigma_n\}$, onde os sujeitos são variáveis, e que seja consistente, ou seja, que nenhuma variável seja sujeito de mais de uma designação de tipo.

3. Uma designação $M : \sigma$ é derivável do contexto Γ, denotado por $\Gamma \vdash M : \sigma$, se $\Gamma \vdash M : \sigma$ pode ser construído usando as regras apresentadas abaixo:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ (Ax)$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.M) : \sigma \to \tau}\ (\to I)$$

$$\frac{\Gamma \vdash M : (\sigma \to \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}\ (\to E)$$

Damos a seguir alguns exemplos típicos de designação de tipos à la Church. Por conveniência e uma boa visualização, às vezes usamos x^σ em vez de $x : \sigma$.

Exemplos 2.2.3: Seja $\sigma, \tau \in \mathbb{T}$. Então:

1. $$\frac{x : \sigma \vdash \sigma}{\vdash \lambda x^\sigma.x : \sigma \to \sigma}\ (\to I)$$

2. $$\frac{x : \sigma, y : \tau \vdash y : \tau}{y : \tau \vdash \lambda x^\sigma.y : \sigma \to \tau}\ (\to I)$$

3. $$\frac{\dfrac{x : \sigma, y : \sigma \vdash x : \sigma}{y : \sigma \vdash \lambda x^\sigma.x : \sigma \to \sigma}\ (\to I) \qquad y : \sigma \vdash y : \sigma}{y : \sigma \vdash (\lambda x^\sigma.x)y : \sigma}\ (\to E)$$

2.2.4 Tipos à la Church versus à la Curry

Apesar das diferenças já mencionadas entre as tipagens à la Church e à la Curry, existe uma conexão entre eles, ou seja, existe uma correspondência que nos permite mudar de um para outro. Esta correspondência, chamada de *apagar*, nos permite mudar da tipagem à la Church para a tipagem à la Curry simplesmente apagando todas as informações de tipos, vejamos:

Definição 2.2.8: A aplicação *apagar* $|\cdot|$ é definida da seguinte forma:

$$|x| = x;$$
$$|MN| = |M|\ |N|;$$
$$|\lambda x^\sigma.M| = \lambda x.|M|.$$

Por exemplo, $|\lambda x^{\sigma\to\tau\to\rho}.\lambda y^{\sigma\to\tau}.\lambda z^{\sigma}.xz(yz)| = \lambda x.\lambda y.\lambda z.xz(yz)$.

Lema 2.2.9: Sejam $M, N \in \Lambda_{\mathbb{T}}$.

(i) Se $M \to_\beta^* N$, então $|M| \to_\beta^* |N|$;

(ii) Se $\Gamma \vdash M : \sigma$ à la Church, então $\Gamma \vdash |M| : \sigma$ à la Curry.

Demonstração: (i) segue por indução em M, e (ii) segue por indução na derivação de $\Gamma \vdash M : \sigma$. ∎

O lema acima nos diz como passar do sistema à la Church para o sistema à la Curry, e o lema abaixo nos diz como fazer o oposto.

Lema 2.2.10: Sejam M, N λ-termos (livre de tipos).

(i) Se $M \to_\beta^* N$ e $M = |M'|$, então $M' \to_\beta^* N'$ para algum $N' \in \Lambda_{\mathbb{T}}$ tal que $N = |N'|$;

(ii) Se $\Gamma \vdash M : \sigma$ à la Curry, então existe $M' \in \Lambda_{\mathbb{T}}$ com $|M'| = M$ e $\Gamma \vdash M' : \sigma$ à la Church.

Demonstração: Indução nas derivações de $M \to_\beta^* N$ e $\Gamma \vdash M : \sigma$ respectivamente. ∎

Os lemas acima nos permitem transladar várias propriedades da tipagem à la Church para os análogos na tipagem à la Curry, e vice-versa. Por exemplo, em ambos os sistemas as seguintes propriedades, dentre outras, são válidas:

Lema da Geração:

(a) $\Gamma \vdash x : \sigma \Rightarrow x : \sigma \in \Gamma$.

(b) $\Gamma \vdash MN : \sigma$ implica que existe um τ tal que $\Gamma \vdash M : \tau \to \sigma$ e $\Gamma \vdash N : \tau$.

(c) $\Gamma \vdash \lambda x.M : \sigma$ implica que existem τ e ρ tal que $\Gamma, x : \tau \vdash M : \rho$ e $\sigma = \tau \to \rho$.

Lema de Redução do Sujeito: Se $\Gamma \vdash M : \sigma$ e $M \to_\beta^* N$, então $\Gamma \vdash N : \sigma$.

Propriedade de Church-Rosser: Suponha que $\Gamma \vdash M : \sigma$.

Se $M \rightarrow_\beta^* N_1$ e $M \rightarrow_\beta^* N_2$, então existe um termo L tal que $N_1 \rightarrow_\beta^* L$, $N_2 \rightarrow_\beta^* L$ e $\Gamma \vdash L : \sigma$.

Normalização: Se $\Gamma \vdash M : \sigma$ então M possui uma forma normal.

Em termos de propriedades, a tipagem à la Church difere da tipagem à la Curry somente no fato de que na tipagem à la Church o tipo de um termo é único, ou seja,

Lema 2.2.11: Se $\Gamma \vdash M : \sigma$ e $\Gamma \vdash M : \tau$, então $\sigma = \tau$.

Demonstração: Indução na estrutura do termo M. ∎

Por exemplo, na tipagem à la Church $\lambda x^\sigma.\lambda y^{(\tau\rightarrow\sigma)\rightarrow\sigma}.y(\lambda z^\tau.x)$ possui somente o tipo $\sigma \rightarrow ((\tau \rightarrow \sigma) \rightarrow \sigma) \rightarrow \sigma$. Enquanto que na tipagem à la Curry $\lambda x.\lambda y.y(\lambda z.x)$ pode ter como designação os tipos $\sigma \rightarrow ((\tau \rightarrow \sigma) \rightarrow \sigma) \rightarrow \sigma$, $\sigma \rightarrow ((\tau \rightarrow \sigma) \rightarrow \rho) \rightarrow \rho$, $(\sigma \rightarrow \sigma) \rightarrow ((\tau \rightarrow \sigma \rightarrow \sigma) \rightarrow \rho) \rightarrow \rho, \cdots$.

Finalizamos este capítulo apresentando o *Teorema do Isomorfismo de Curry-Howard*. Como observado por Morten em [67], este isomorfismo estabelece uma conexão entre *Teoria da Prova* e *Teoria dos Tipos*; por exemplo, *a lógica intuicionista minimal* corresponde ao λ-cálculo tipado, a *lógica de primeira ordem* corresponde aos *tipos dependentes*, a *lógica de segunda ordem* corresponde aos *tipos polimórficos*, o cálculo de sequentes é relacionado com a *substituição implícita*, e assim por diante.

Teorema do Isomorfismo de Curry-Howard:

(i) Se $\Gamma \vdash M : \varphi$ então $||\Gamma|| \vdash \varphi$

(ii) Se $\Sigma \vdash \varphi$ então existe um termo tipado M tal que $\Gamma \vdash M : \varphi$, onde $\Gamma = \{(x_\varphi : \varphi) \mid \varphi \in \Sigma\}$.

Demonstração: (i) por indução na derivação de $\Gamma \vdash M : \varphi$, e (ii) por indução na derivação de $\Sigma \vdash \varphi$. Para mais detalhes ver [67]. ∎

Capítulo 3

O PVS e sua Semântica: Uma Visão Geral

Desenvolvido pela SRI International, o PVS[1] (Prototype Verification System [14]) é um provador interativo de teoremas que combina uma expressiva *Linguagem de Especificação* (ver Apêndice A) com um *Assistente de Prova Interativo* (ver Apêndice B), e vem sendo muito utilizado para especificar e verificar propriedades de *hardware* [65], protocolos [28, 31], propriedades de sistemas tempo-real [30, 65] e sistemas críticos, como por exemplo, detecção e resolução de conflitos no tráfego aéreo [22]. De certa forma, o PVS foi construído para agregar uma combinação de técnicas já desenvolvidas para outros sistemas e mostradas serem extremamente úteis. Por exemplo, os sistemas Nuprl [12] e Veritas [26] inspiraram a construção dos subtipos e dos tipos dependentes; as técnicas do assistente de prova foram baseadas no LCF [25] e no Boyer-Moore [9]; enquanto que as *estratégias*, o que permite ao usuário um certo nível de automação nas provas, são equivalentes às *tacticals* em HOL.

Como vimos no Capítulo 2, já é um fato que os tipos exercem um papel importante na teoria da computação e é melhor exemplificado pelo seu uso em várias linguagens de programação tal como Algol, Ada, e ML, e é também pesadamente enfatizado no PVS. O PVS consiste, dentre outras características, de um sistema de tipos baseado na Teoria de Tipos Simples de Church (Ver Seção 2.2.3), porém, estendida com *subtipos*, *tipos dependentes* e *data types*. Dentre os tipos básicos destacamos `bool`, `nat`, `rational` e `real`. Tipos mais complexos podem ser construídos usando *construtores de tipos*. Alguns destes contrutores são usados para construir tipos *funcionais*, *registros* e *produtos*, por exemplo,

[1] Disponível em `http://pvs.csl.sri.com/`.

um *tipo funcional* de *tipo domínio* `T1` para o *tipo imagem* `T2` é construído como `[T1 -> T2]`, e o *tipo produto* de `T1`, `T2` é construído como `[T1, T2]`. Por outro lado, os *tipos registro* são da forma `[# a1:T1, ..., an:Tn #]` onde os $\mathtt{a}_i$ são os nomes das diferentes componentes do registro e os $\mathtt{T}_i$ são expressões de tipo.

A seguir damos somente uma visão geral da semântica do PVS. Para mais detalhes e demonstrações do resultados apresentados aqui veja o texto [51] do próprio PVS que discorre sobre a sua semântica formal, e que é a base para todo esse capítulo.

3.1 Contextos e Checagem de Tipo em PVS

É sabido que os tipos também servem como um mecanismo poderoso para detectar erros *sintáticos* e *semânticos*. Esta verificação é feita através da *checagem de tipos* usando a *operação typechecking*, a qual determina se uma dada expressão esta *bem-tipada* com respeito a um dado *contexto*. Em PVS o *typechecking* é definido como uma função parcial τ que designa um tipo para o termo a com respeito a um *contexto* Γ.

Um *contexto* Γ, em PVS, é visto como uma função parcial que designa uma *espécie* (ou `TYPE`, ou `CONSTANT`, ou `VARIABLE`) para cada símbolo, e um *tipo* para cada símbolo de constante e variável. Em outras palavras, um contexto em PVS é uma sequência de *declarações*, onde cada declaração é uma declaração de tipo `T:TYPE`, ou de constante `c:T` onde `T` é um tipo, ou de variável `x:VAR T`. De maneira mais formal, um contexto, sendo uma função parcial, pode ser aplicado a um símbolo, de acordo com a seguinte definição:

Definição 3.1.1: Sejam Γ um contexto, s e r declarações, e $r \neq s$. Sendo s uma declaração D e $\Gamma' = \Gamma \cup \{s : D\}$ um contexto, ou, como definido na Seção 2.1.1, $\Gamma' = \{\Gamma, s : D\}$, então:

(a) $(\Gamma')(s) = D$;

(b) $(\Gamma')(r) = \Gamma(r)$;

(c) se r não esta declarado no contexto Γ, então $\Gamma(r)$ é dito ser *indefinido*;

(d) a *espécie* do símbolo s em Γ' é dada por $kind(\Gamma'(s))$;

(e) se a espécie de s é `CONSTANT` ou `VARIABLE`, então a *designação de tipo* para s em Γ' é dada por $type(\Gamma(s))$.

O exemplo a seguir deixa bem claro o uso dos conceitos apresentados na definição acima.

Exemplo 3.1.1: A sequência de declarações a seguir é um contexto, e o chamamos de Γ.

$$\texttt{bool} : \texttt{TYPE}, \quad \texttt{TRUE} : \texttt{bool}, \quad \texttt{FALSE} : \texttt{bool}, \quad x\text{: } \texttt{VAR}\,[[\texttt{bool}, \texttt{bool}] \to \texttt{bool}]$$

Neste contexto temos:

$kind(\Gamma(\texttt{bool}))$	=	`TYPE`	$type(\Gamma(\texttt{bool}))$	=	`TYPE`
$kind(\Gamma(\texttt{TRUE}))$	=	`CONSTANT`	$type(\Gamma(\texttt{FALSE}))$	=	`bool`
$kind(\Gamma(x))$	=	`VARIABLE`	$type(\Gamma(x))$	=	$[[\texttt{bool}, \texttt{bool}] \to \texttt{bool}]$

A definição a seguir, a qual apresenta a *operação typechecking* que verifica a boa-formação de contextos, tipos e termos, onde os termos (Veja Seção 2.2.1) podem ser constantes (c, f), variáveis (x), pares $((a, b))$, projeções $(p_i\ a)$, aplicações $(f\ a)$, ou abstrações $(\lambda(x : T) : a)$, nos diz que:

(1) Um contexto Γ é checado com respeito a um contexto vazio, representado por `{ }`, ou melhor, $\tau()(\Gamma)$ retorna `CONTEXT` quando Γ é um contexto bem-formado;

(2) Para uma expressão A, $\tau(\Gamma)(A)$ retorna `TYPE` quando A é bem-formado com respeito ao contexto Γ;

(3) O resultado da aplicação $\tau(\Gamma)(a)$ sobre um termo bem-formado a é o tipo que é designado para a com relação ao contexto Γ, o qual é chamado de *tipo canônico*.

Definição 3.1.2 [Operação Typechecking]:

$$
\begin{array}{rcl}
\tau()(\{\}) & = & \texttt{CONTEXT} \\
\tau()(\Gamma, s:\ \texttt{TYPE}) & = & \texttt{CONTEXT}, \text{ se } \Gamma(s) \text{ é indefinido} \\
 & & \text{e } \tau()(\Gamma) = \texttt{CONTEXT} \\
\tau()(\Gamma, c:T) & = & \texttt{CONTEXT}, \text{ se } \Gamma(c) \text{ é indefinido, } \tau(\Gamma)(T) = \texttt{TYPE} \\
 & & \text{e } \tau()(\Gamma) = \texttt{CONTEXT} \\
\tau()(\Gamma, x:\texttt{VAR}\,T) & = & \texttt{CONTEXT}, \text{ se } \Gamma(x) \text{ é indefinido,} \\
 & & \tau(\Gamma)(T) = \texttt{TYPE} \text{ e } \tau()(\Gamma) = \texttt{CONTEXT} \\
 & & \\
\tau(\Gamma)(s) & = & \texttt{TYPE} \text{ se } \mathit{kind}(\Gamma(s)) = \texttt{TYPE} \\
\tau(\Gamma)([A \to B]) & = & \texttt{TYPE} \text{ se } \tau(\Gamma)(A) = \tau(\Gamma)(B) = \texttt{TYPE} \\
\tau(\Gamma)([A_1, A_2]) & = & \texttt{TYPE} \text{ se } \tau(\Gamma)(A_i) = \texttt{TYPE} \text{ para } 1 \leq i \leq 2 \\
 & & \\
\tau(\Gamma)(a) & = & \mathit{type}(\Gamma(a)) \\
 & & \text{se } \mathit{kind}(\Gamma(a)) \in \{\texttt{CONSTANT}, \texttt{VARIABLE}\} \\
\tau(\Gamma)(f\ a) & = & B \text{ se } \tau(\Gamma)(f) = [A \to B] \text{ e } \tau(\Gamma)(a) = A \\
\tau(\Gamma)(\lambda(x:T):a) & = & [T \to \tau(\Gamma, x:\texttt{VAR}\,T)(a)] \text{ se } \Gamma(x) \text{ é indefinido} \\
 & & \text{e } \tau(\Gamma)(T) = \texttt{TYPE} \\
\tau(\Gamma)((a_1, a_2)) & = & [\tau(\Gamma)(a_1), \tau(\Gamma)(a_2)] \\
\tau(\Gamma)(p_i\ a) & = & T_i \text{ onde} \\
 & & \tau(\Gamma)(a) = [T_1, T_2]
\end{array}
$$

Na regra para λ-abstração a restrição de que $\Gamma(x)$ deve ser indefinido pode ser satisfeita renomeando convenientemente a variável ligada. Além disso, pela própria definição recursiva das regras para contexto é fácil ver que vale o seguinte lema:

Lema 3.1.3: A *operação typechecking* quando aplicada a um contexto, preserva a boa-formação do contexto em cada chamada recursiva, de modo que se o contexto inicial é bem-formado, então assim o é todo contexto intermediário.

Exemplo 3.1.2: Seja $\Omega = \{\texttt{bool} : \texttt{TYPE},\ \texttt{TRUE} : \texttt{bool},\ \texttt{FALSE} : \texttt{bool}\}$ um contexto.

(a) $\tau()(\{\}) = \texttt{CONTEXT}$

(b) $\tau()(\Omega) = \texttt{CONTEXT}$

(c) $\tau(\Omega)([[\texttt{bool}, \texttt{bool}] \to \texttt{bool}]) = \texttt{TYPE}$

(d) $\tau(\Omega)((\texttt{TRUE}, \texttt{FALSE})) = [\texttt{bool}, \texttt{bool}]$

(e) $\tau(\Omega)(p_2\ (\texttt{TRUE}, \texttt{FALSE})) = \texttt{bool}$

(f) $\tau(\Omega)(\lambda(x : \texttt{bool}) : \texttt{TRUE}) = [\texttt{bool} \rightarrow \texttt{bool}]$

É fácil verificar as afirmações acima pela definição. Mas antes de iniciar a verificação observe que, por definição, $\tau(\Omega)(\texttt{bool}) = \texttt{TYPE}$, $\tau(\Omega)(\texttt{TRUE}) = \texttt{bool}$ e $\tau(\Omega)(\texttt{FALSE}) = \texttt{bool}$, pois, $kind(\Omega(\texttt{bool})) = \texttt{TYPE}$, $kind(\Omega(\texttt{TRUE})) = \texttt{bool}$ e $kind(\Omega(\texttt{FALSE})) = \texttt{bool}$. Então,

(a) $\tau()(\{\}) = \texttt{CONTEXT}$;

(b) $\tau()(\Omega) = \texttt{CONTEXT}$. De fato, $\tau()(\{\}, \texttt{bool} : \texttt{TYPE}) = \texttt{CONTEXT}$, uma vez que $\{\}(\texttt{bool})$ é indefinido e $\tau()(\{\}) = \texttt{CONTEXT}$;

Sendo o contexto $\Omega_1 = \{\{\}, \texttt{bool} : \texttt{TYPE}\}$, ou seja, $\Omega_1 = \{\texttt{bool} : \texttt{TYPE}\}$, vem que $\tau()(\Omega_1, \texttt{TRUE} : \texttt{bool}) = \texttt{CONTEXT}$, pois $\Omega_1(\texttt{TRUE})$ é indefinido, $\tau(\Omega_1)(\texttt{bool}) = \texttt{TYPE}$ e $\tau()(\Omega_1) = \texttt{CONTEXT}$;

Seja o contexto $\Omega_2 = \{\texttt{bool} : \texttt{TYPE}, \texttt{TRUE} : \texttt{bool}\}$, ou seja, $\Omega = \{\Omega_2, \texttt{FALSE} : \texttt{bool}\}$. Assim, temos que $\tau()(\Omega) = \tau()(\Omega_2, \texttt{FALSE} : \texttt{bool}) = \texttt{CONTEXT}$, pois $\Omega_2(\texttt{FALSE})$ é indefinido, $\tau(\Omega_1)(\texttt{bool}) = \texttt{TYPE}$ e $\tau()(\Omega_2) = \texttt{CONTEXT}$;

(c) $\tau(\Omega)([\texttt{bool}, \texttt{bool}]) = \texttt{TYPE}$, portanto, $\tau(\Omega)([[\texttt{bool}, \texttt{bool}] \rightarrow \texttt{bool}]) = \texttt{TYPE}$;

(d) $\tau(\Omega)((\texttt{TRUE}, \texttt{FALSE})) = [\tau(\Omega)(\texttt{TRUE}), \tau(\Omega)(\texttt{FALSE})] = [\texttt{bool}, \texttt{bool}]$;

(e) $\tau(\Omega)(p_2\ (\texttt{TRUE}, \texttt{FALSE})) = \texttt{bool}$, pois, $\tau(\Omega)((\texttt{TRUE}, \texttt{FALSE})) = [\texttt{bool}, \texttt{bool}]$.

(f) Pela definição temos que:

$$\tau(\Omega)(\lambda(x : \texttt{bool}) : \texttt{TRUE}) = [\texttt{bool} \rightarrow \tau(\Omega, x : \texttt{VAR bool})(\texttt{TRUE})]$$

se $\Omega(x)$ é indefinido e $\tau(\Omega)(\texttt{bool}) = \texttt{TYPE}$.

Como TRUE é uma constante vem que

$$\tau(\Omega, x : \texttt{VAR bool})(\texttt{TRUE}) = type(\Omega(\texttt{TRUE}) = \texttt{bool}.$$

Uma vez que $\Omega(x)$ é indefinido e $\tau(\Omega)(\texttt{bool}) = \texttt{TYPE}$ obtemos que,

$$\tau(\Omega)(\lambda(x : \texttt{bool}) : \texttt{TRUE}) = [\texttt{bool} \rightarrow \texttt{bool}].$$

A próxima proposição nos diz que quando estendemos um contexto os juízos (Ver Capítulo 2) de tipo não são invalidados.

Proposição 3.1.4: Se $\tau()(\Gamma) = \tau()(\Gamma') = \texttt{CONTEXT}$ e Γ é um prefixo de Γ', então para todo tipo A, $\tau(\Gamma)(A) = \texttt{TYPE}$ implica $\tau(\Gamma')(A) = \texttt{TYPE}$, e para todo termo a, $\tau(\Gamma)(a) = A$ implica $\tau(\Gamma')(a) = A$.

Demonstração: A prova desta proposição segue diretamente da Definição 3.1.2 e do Lema 3.1.3. Como exemplo, daremos um *sketch* da prova do caso $\tau(\Gamma)(A) = \texttt{TYPE}$ implica $\tau(\Gamma')(A) = \texttt{TYPE}$. Uma vez que $\tau(\Gamma)(A) = \texttt{TYPE}$, por definição, temos que $kind(\Gamma(A)) = \texttt{TYPE}$ o que nos leva a concluir que $\Gamma(A)$ não é indefinido, ou seja, $A : \texttt{TYPE}$ esta declarado no contexto Γ. Como $\Gamma \subseteq \Gamma'$ temos que $A : \texttt{TYPE}$ esta declarado no contexto Γ'. Sendo Γ' um contexto bem-formado, pelo Lema 3.1.3, vem que $\Gamma' - \{A\}$ também é um contexto bem-formado. Portanto, como $kind(\Gamma'(A)) = \texttt{TYPE}$, concluimos que $\tau(\Gamma')(A) = \texttt{TYPE}$. ■

Os contextos em PVS ainda são enriquecidos de modo que podem conter declarações de tipos da forma $s : \texttt{TYPE} = T$, onde T é um tipo. Neste caso, se o contexto Γ contém tal declaração para s, então definimos $definition(\Gamma(s))$ como sendo T. Sendo assim, inevitavelmente devemos aumentar o conjunto de regras da *operação typechecking* para lidar com tal situação. Isto é feito acrescentando à Definição 3.1.2 as seguintes regras:

$$\begin{aligned}
\tau()(\Gamma, s : \texttt{TYPE} = T) &= \texttt{CONTEXT}, \text{ se } \Gamma(s) \text{ é indefinido,} \\
&\quad \tau()(\Gamma) = \texttt{CONTEXT} \\
&\quad \text{e } \tau(\Gamma)(T) = \texttt{TYPE} \\
\\
\tau(\Gamma)(s) &= \delta(\Gamma)(type(\Gamma(s))) \\
&\quad \text{se } kind(\Gamma(s)) \in \{\texttt{CONSTANT}, \texttt{VARIABLE}\}
\end{aligned}$$

onde $\delta(\Gamma)(T)$ é a operação que retorna a forma expandida de um tipo relativo ao contexto Γ, e é definido como segue

Definição 3.1.5 [Tipo Expandido]:

$$\begin{aligned}
\delta(\Gamma)(s) &= s, \text{ se } definition(\Gamma(s)) \text{ é vazio} \\
\\
\delta(\Gamma)(s) &= \delta(\Gamma)(definition(\Gamma(s))), \text{ se } definition(\Gamma(s)) \text{ é não-vazio}
\end{aligned}$$

$$\begin{aligned}\delta(\Gamma)([A \to B]) &= [\delta(\Gamma)(A) \to \delta(\Gamma)(B)] \\ \delta(\Gamma)([T_1, T_2]) &= [\delta(\Gamma)(T_1), \delta(\Gamma)(T_2)]\end{aligned}$$

Note que o operador δ é indepotente, e que $\tau(\Gamma)(a)$, para um certo termo a, sempre retorna um tipo expandido, ou seja, $\delta(\Gamma)(\tau(\Gamma)(a)) = \tau(\Gamma)(a)$.

Exemplo 3.1.3: Considere o conjunto de declarações

$$\Omega' = \{\Omega, \texttt{boolp} : \texttt{TYPE} = [[\texttt{bool}, \texttt{bool}] \to \texttt{bool}], V : \texttt{boolop}\},$$

onde Ω é tal como apresentado na Definição 3.1.2. Então

$$\begin{aligned}\tau()(\Omega') &= \texttt{CONTEXT} \\ \tau(\Omega')(\texttt{boolop}) &= [[\texttt{bool}, \texttt{bool}] \to \texttt{bool}] \\ \tau(\Omega')(V) &= [[\texttt{bool}, \texttt{bool}] \to \texttt{bool}]\end{aligned}$$

3.2 Os Subtipos em PVS

Do ponto de vista de subtipos, em PVS, é possível introduzir o "tipo" números naturais como um "subtipo" do "tipo" números reais, e tratar os primos, os números pares, e os números ímpares como subtipos do tipo números naturais, assim como, introduzem a possibilidade de tipos serem vazios; isto porque, em PVS, os subtipos correspondem na teoria dos conjuntos à noção de subconjuntos.

De maneira geral, em PVS, o *subtipo* (ou *subtipo predicado*) dos elementos de um tipo T satisfazendo o predicado P é definido como sendo o tipo $\{x : T \mid P(x)\}$, onde P é definido como sendo um tipo funcional cujo tipo imagem é o tipo primitivo `bool` e $P(x)$ é qualquer fórmula em PVS. Além disso, subtipos sobre tipos funcionais são contravariantes no tipo imagem, mas não são contravariantes sobre o tipo domínio, ou seja, não podemos ver o tipo funcional `[nat -> nat]` como um subtipo do tipo funcional `[int -> nat]`. Tal relação de subtipo violaria a *extensionalidade*[2]. Em outras palavras, o tipo funcional $[A \to B]$ é um subtipo de $[A' \to B']$ se, e somente se, B é um subtipo de B', e A e A'

[2]Duas funções sobre os números naturais são extencionalmente iguais quando elas retornam valores iguais se aplicadas em iguais argumentos naturais.

são *tipos equivalentes*. Como A e A' podem ser subtipos gerados por predicados, esta equivalência se reduz a verificar a equivalência entre tais predicados. A equivalência de tipos citada será definida mais adiante e denotada por $\simeq$.

Novamente, uma vez que foi adicionado os subtipos à linguagem do PVS, se faz necessário atualizar a *operação typechecking*. No entanto, para fazer esta atualização são necessários os seguintes operadores:

- $\mu(A)$: Este operador, definido a seguir, retorna o *supertipo maximal* de um tipo A, onde um *tipo maximal* s é um tipo tal que $\mu(s) = s$.

$$\begin{array}{rcl} \mu(s) & = & s \\ \mu(\{x : T \mid a\}) & = & \mu(T) \\ \mu([A \to B]) & = & [A \to \mu(B)] \\ \mu([A_1, A_2]) & = & [\mu(A_1), \mu(A_2)] \end{array}$$

- $\pi(A)$: Este operador retorna os predicados que restringem um tipo A relativo ao seu supertipo maximal $\mu(A)$, e é definido como segue:

$$\begin{array}{rcl} \pi(s) & = & \lambda(x : s) : \texttt{TRUE} \\ \pi(\{y : T \mid a\}) & = & \lambda(x : \mu(T)) : (\pi(T)(x) \wedge a[x/y]) \\ \pi([A \to B]) & = & \lambda(x : [A \to \mu(B)]) : (\forall(y : A) : \pi(B)(x(y))) \\ \pi([A_1, A_2]) & = & \lambda(x : [\mu(A_1), \mu(A_2)]) : (\pi(A_1)(p_1\ x) \wedge \pi(A_2)(p_2\ x) \end{array}$$

  onde $a[x/y]$ significa que x é substituído por y em a. Esta operação de *substituição* é realizada de acordo com a Definição 3.2.1 apresentada mais adiante.

- $\mu_0(A)$: Este operador é uma variante do operador μ e retorna o *supertipo direto* o qual considera somente os supertipos de subtipos dados explicitamente na forma $\{x : T \mid a\}$. Vejamos a sua definição:

$$\begin{array}{rcl} \mu(\{x : T \mid a\}) & = & \mu(T) \\ \mu(T) & = & T, \text{ caso contrário} \end{array}$$

Definição 3.2.1 [Substituição]:

$$
\begin{array}{rcl}
s[a_1/x_1,\cdots,a_n/x_n] & = & \left\{\begin{array}{ll} a_i, & \text{se para algum minimal } i,\ s \equiv x_i \\ s, & \text{caso contrário} \end{array}\right. \\
(f\ \ a)[a_1/x_1,\cdots,a_n/x_n] & = & (f[a_1/x_1,\cdots,a_n/x_n] \quad a[a_1/x_1,\cdots,a_n/x_n]) \\
(\lambda(y:T):a)[a_1/x_1,\cdots,a_n/x_n] & = & (\lambda(y':T):\ \ a[y'/y,a_1/x_1,\cdots,a_n/x_n]) \\
 & & \text{onde } y' \text{ é uma variável fresca} \\
(b_1,b_2)[a_1/x_1,\cdots,a_n/x_n] & = & (b_1[a_1/x_1,\cdots,a_n/x_n]\ ,\ \ b_2[a_1/x_1,\cdots,a_n/x_n]) \\
(p_i\ \ a)[a_1/x_1,\cdots,a_n/x_n] & = & (p_i\ \ a[a_1/x_1,\cdots,a_n/x_n])
\end{array}
$$

Exemplo 3.2.1: Considere o seguinte contexto.

$$
\begin{array}{rcl}
\texttt{int} & : & \texttt{TYPE} \\
\texttt{0} & : & \texttt{int} \\
\leq & : & [[\texttt{int},\texttt{int}] \to \texttt{bool}] \\
\texttt{nat} & : & \texttt{TYPE} = \{\texttt{i} : \texttt{int} \mid \texttt{0} \leq \texttt{i}\} \\
\texttt{natinjection} & : & \texttt{TYPE} = \{\texttt{f} : [\texttt{nat} \to \texttt{nat}] \mid \forall(\texttt{i},\texttt{j}:\texttt{nat}) : \texttt{f(i)} = \texttt{f(j)} \supset \texttt{i} = \texttt{j}\}
\end{array}
$$

Dessa forma temos:

$$
\begin{array}{rcl}
\mu(natinjection) & = & \mu([\texttt{nat} \to \texttt{nat}]) \\
 & = & [\texttt{nat} \to \mu(\texttt{nat})] \\
 & = & [\texttt{nat} \to \texttt{int}] \\
 & & \\
\mu_0(\texttt{natinjection}) & = & [\texttt{nat} \to \texttt{nat}] \\
 & & \\
\pi(\texttt{natinjection}) & = & \lambda(\texttt{f} : [\texttt{nat} \to \texttt{int}]) : \quad \pi([\texttt{nat} \to \texttt{nat}])(\texttt{f}) \quad \wedge \\
 & & \qquad\qquad (\forall(\texttt{i},\texttt{j}:\texttt{nat}) : \texttt{f(i)} = \texttt{f(j)} \supset \texttt{i} = \texttt{j})
\end{array}
$$

onde

$$
\pi([\texttt{nat} \to \texttt{nat}]) \quad = \quad \lambda(\texttt{g} : [\texttt{nat} \to \texttt{int}]) : \forall(\texttt{i} : \texttt{nat}) : (\lambda(\texttt{j} : \texttt{int}) : \texttt{0} \leq \texttt{j})(\texttt{g(i)}))
$$

Como mencionamos antes, uma definição para estabelecer que dois tipos são equivalentes se faz necessário, pois, por exemplo, o subtipo $\{x : T \mid p(x) \wedge q(x)\}$ pode também ser escrito como $\{x : T \mid q(x) \wedge p(x)\}$. Esta equivalência, denotada por $\simeq$ e definida a seguir, retorna uma lista das obrigações de prova que devem ser demonstradas, e é aplicada somente a tipos maximais. Além disso, a definição de $\simeq$ faz uso do predicado de igualdade do PVS o qual é definido por $= : [[T, T] \to \texttt{bool}]$ onde T é um tipo, e é usado

o símbolo ";" para denotar a concatenação de duas listas.

Definição 3.2.2 [Equivalência de Tipos]:

$$
\begin{aligned}
(s \simeq s) &= \texttt{TRUE} \\
([A \to B] \simeq [A' \to B']) &= ((\mu(A) \simeq \mu(A')); (\pi(A) = \pi(A')); (B \simeq B')) \\
([A_1, A_2] \simeq [B_1, B_2]) &= ((A_1 \simeq B_1); (A_2 \simeq B_n)) \\
(A \simeq B) &= \texttt{FALSE}, \text{ caso contrário.}
\end{aligned}
$$

Agora estamos na eminência de acrescentar à Definição 3.1.2 as novas regras necessárias para lidar com os subtipos. Porém, temos ainda que introduzir um outro conceito extremamente importante que é a *compatibilidade* entre tipos. Vejamos com um exemplo o porque de introduzir formalmente este conceito: Suponhamos que temos uma aplicação $(f\ \ a)$ onde o tipo de f é $[A \to B]$ e o tipo de a é A'. Neste caso, devemos garantir que o termo a satisfaz o predicado imposto pelo tipo (subtipo) A, ou seja, devemos garantir que os tipos A e A' sejam compatíveis.

Definição 3.2.3 [Compatibilidade]: Dois tipos A e B são compatíveis em um contexto Γ, e denotado por $(A \sim B)_\Gamma$, se, e somente se, $(\mu(A) \simeq \mu(B))$.

Finalmente, apresentamos a seguir as regras que devem se juntar com a Definição 3.1.2 para contemplar o caso de subtipos.

$$
\begin{aligned}
\tau()(\{\Gamma, c : T\}) &= \texttt{CONTEXT} \text{ se } \Gamma(c) \text{ é indefinido,} \\
&\quad \tau(\Gamma)(T) = \texttt{TYPE}, \\
&\quad \tau()(\Gamma) = \texttt{CONTEXT}, \text{ e} \\
&\quad \vdash_\Gamma (\exists (x : T) : \texttt{TRUE}) \\
\\
\tau(\Gamma)(x : T \mid a) &= \texttt{TYPE}, \text{ se } \Gamma(x) \text{ é indefinido,} \\
&\quad \tau(\Gamma)(T) = \texttt{TYPE}, \text{ e } \tau(\Gamma, x : VAR\, T)(a) = \texttt{bool} \\
\\
\tau(\Gamma)(f\ \ a) &= B, \text{ onde } \mu_0(\tau(\Gamma)(f)) = [A \to B], \\
&\quad \tau(\Gamma)(a) = A', \\
&\quad (A \sim A')_\Gamma, \\
&\quad \vdash_\Gamma \pi(A)(a) \\
\\
\tau(\Gamma)(p_i\ \ a) &= A_i, \text{ onde } \mu_0(\tau(\Gamma)(a)) = [A_1, A_2]
\end{aligned}
$$

Exemplo 3.2.2: Considerando e denotando o contexto dado no exemplo anterior de Γ, temos que $\tau(\Gamma)(\{\mathtt{i} : \mathtt{int} \mid \mathtt{0} \leq \mathtt{i}\}) = \mathtt{TYPE}$, pois $\Gamma(\mathtt{i})$ é indefinido, $\tau(\Gamma)(\mathtt{int}) = \mathtt{TYPE}$, e $\tau(\Gamma, \mathtt{i} : \mathtt{VAR\ int})(\mathtt{0} \leq \mathtt{i}) = \mathtt{bool}$.

3.3 Os Tipos Dependentes em PVS

Além do já mencionado, o que torna o sistema de tipos do PVS bastante expressivo, o sistema de tipos do PVS inclui também *tipos dependentes*. A combinação de tipos dependentes com subtipos predicados fornece um meio extremamente poderoso que permite, por exemplo, capturar as relações entre o tipo imagem e o tipo domínio de um tipo funcional. Porém, a introdução de tipos dependentes no PVS tem suas limitações e preserva uma certa invariância. Mais especificamente, em um tipo dependente $T(n)$, o parâmetro n pode ocorrer somente dentro do predicado que compõe $T(n)$, como por exemplo, no tipo dependente $\mathtt{below(n)} = \{\mathtt{s} : \mathtt{nat} \mid \mathtt{s} < \mathtt{n}\}$. Em outras palavras, a estrutura de $T(n)$ é invariante com respeito a n e, por isso, não se pode definir um construtor de tipos em PVS que retorne uma n-tupla $\underbrace{[A, [\cdots, A]]}_{n}$ sobre o tipo A.

Um *tipo produto dependente* é escrito da forma $[x : A, B]$, enquanto que uma *tipo funcional dependente* é escrito da forma $[x : A \rightarrow B]$, como mostram os exemplos a seguir:

Exemplo 3.3.1:

(a) $[i : nat, \{j : nat \mid j \leq i\}]$

(b) $[i : nat, [\{j : nat \mid j \leq i\} \rightarrow \mathtt{bool}]]$

(c) $[i : int \rightarrow \{j : int \mid i \leq j\}]$

Como já era de se esperar, com a adição de tipos dependentes, são necessárias mudanças nos operadores μ, π, na relação $\simeq$, na *operação typechecking* e na maneira como se faz as substituições, apesar de que em alguns deles a mudança é mínima. Por exemplo, a definição de μ é essencialmente inalterada exceto que para um tipo ligado, $\mu(x : T) = x : \mu(T)$.

No entanto, a definição de π para um tipo funcional dependente $[y : A \to B]$ é ligeiramente diferente da de um tipo funcional ordinário. Enquanto que a definição para tipo produto dependente permanece essencialmente inalterada da de um tipo produto ordinário.

As definições a seguir mostram as mudanças necessárias, respectivamente, para substituições e para o operador π.

Definição 3.3.1 [Substituição para Tipos]:

$$[x : A \to B][a_1/x_1, \cdots, a_n/x_n] = [y : A[a_1/x_1, \cdots, a_n/x_n] \to B[y/x, a_1/x_1, \cdots, a_n/x_n]]$$

$$[x : A, B][a_1/x_1, \cdots, a_n/x_n] = [y : A[a_1/x_1, \cdots, a_n/x_n], B[y/x, a_1/x_1, \cdots, a_n/x_n]]$$

$$\{[x : A \mid a]\}[a_1/x_1, \cdots, a_n/x_n] = \{y : A[a_1/x_1, \cdots, a_n/x_n] \mid a[y/x, a_1/x_1, \cdots, a_n/x_n]\}$$

Definição 3.3.2 [O operador π]:

$$\begin{aligned} \pi([y : A \to B]) &= \lambda(x : [y : A \to \mu(B)]) : (\forall(y : A) : \pi(B)(x(y)))) \\ \pi([y : A, B]) &= \lambda(x : [y : \mu(A), \mu(B)]) : \pi(A)(p_1\ x) \wedge \pi(B)(p_2\ x)[(p_1\ x)/y])) \end{aligned}$$

Exemplo 3.3.2:

$$\begin{aligned} \mu([i : int \to \{j : int \mid i \leq j\}]) &= [i : int \to int] \\ \pi([i : int \to \{j : int \mid i \leq j\}]) &= \lambda(f : [i : int \to int]) : \\ &\qquad \forall(i : int) : (\lambda(j : int) : i \leq j)(f(i)) \end{aligned}$$

A definição de $\simeq$, a qual checa se dois tipos maximais são equivalentes gerando obrigações de prova, também é ligeiramente adaptada para tipos dependentes. Além disso, será permitido a opção de dois tipos maximais, digamos A e B, serem comparados usando $\simeq$ no contexto de uma expressão a como $(A \simeq B)/a$.

Definição 3.3.3 [Equivalência de Tipos para Tipos Dependentes]:

$$\begin{aligned} (s \simeq s)/a &= \texttt{TRUE} \\ ([x : A \to B] \simeq [x' : A' \to B']) &= (\mu(A) \simeq \mu(A')); \\ &\quad (\pi(A) = \pi(A')); \\ &\quad (\forall(x : A) : (B \simeq B'[x/x'])) \end{aligned}$$

$$
\begin{array}{rcl}
([x : A \to B] \simeq [x' : A' \to B'])/a & = & (\mu(A) \simeq \mu(A')); \\
& & (\pi(A) = \pi(A')); \\
& & (\forall(x : A) : (B \simeq B'[x/x'])/a(x)) \\
\\
([x : A_1, A_2] \simeq [y : B_1, B_2]) & = & (A_1 \simeq B_1); \\
& & (\forall(x_1 : A_1) : (A_2 \simeq B_2[x_1/y_1])) \\
\\
([x : A_1, A_2] \simeq [y : B_1, B_2])/a & = & (A_1 \simeq B_1)/(p_1\ a); \\
& & (A_2[(p_1\ a)/x] \simeq B_2[(p_1\ a)/y])/(p_2\ a) \\
\\
(A \simeq B)/a & = & \texttt{FALSE}, \text{ caso contrário.}
\end{array}
$$

Finalmente, apresentamos através da definição a seguir, as modificações necessárias na *operação typechecking* para que esta possa responder adequadamente à qualquer dependência.

Definição 3.3.4 [Operação Typechecking com Tipos Dependentes]:

$$
\begin{array}{rcl}
\tau(\Gamma)([x : A, B]) & = & \texttt{TYPE} \text{ se } \Gamma(x) \text{ é indefinido}, \\
& & \tau(\Gamma)(A) = \texttt{TYPE}, \text{ e} \\
& & \tau(\Gamma, x : \texttt{VAR}\ A)(B) = \texttt{TYPE} \\
\\
\tau(\Gamma)([x : A \to B]) & = & \texttt{TYPE} \text{ se } \Gamma(x) \text{ é indefinido}, \\
& & \tau(\Gamma)(A) = \texttt{TYPE}, \text{ e} \\
& & \tau(\Gamma, x : \texttt{VAR}\ A)(B) = \texttt{TYPE} \\
\\
\tau(\Gamma)(f\ a) & = & B', \text{ onde } \tau(\Gamma)(f) = [x : A \to B], \\
& & \tau(\Gamma)(a) = A', \\
& & (A \overset{a}{\sim} A')_\Gamma, \\
& & B' \text{ é } B[a/x], \\
& & \vdash_\Gamma \pi(A)(a) \\
\\
\tau(\Gamma)(\lambda(x : A) : a) & = & [x : A \to B] \text{ onde} \\
& & B = \tau(\Gamma, x : \texttt{VAR}\ A)(a) \\
\\
\tau(\Gamma)(p_1\ a) & = & A_1 \text{ onde } \mu_0(\tau(\Gamma)(a)) = [x : A_1, A_2] \\
\\
\tau(\Gamma)(p_2\ a) & = & A_2[(p_1\ a)/x] \text{ onde } \mu_0(\tau(\Gamma)(a)) = [x : A_1, A_2]
\end{array}
$$

Como $(A \sim B)_\Gamma$, a notação $(A \overset{a}{\sim} B)_\Gamma$ indica que todas as obrigações a em $(\mu(A) \simeq \mu(B))/a$ podem ser provadas.

Exemplo 3.3.3:

$$\tau(\Gamma)([x : \texttt{bool}, \{y : \texttt{bool} \mid x \supset y\}]) = \texttt{TYPE}$$

$$\tau(\Gamma)([x : \texttt{bool} \rightarrow \{y : \texttt{bool} \mid x \supset y\}]) = \texttt{TYPE}$$

3.4 O PVS e suas *teorias*

Uma *teoria* em PVS é construída de modo a fornecer uma coleção de declarações, permitindo polimorfismo, e podendo ou não conter parâmetros. Para se ter uma idéia mais prática sobre *teorias* veja o Apêndice A.

3.4.1 *teorias* não-Parametrizadas

Iniciamos com *teorias* sem parâmetros as quais possuem a forma $m : \texttt{THEORY} = \Delta$, onde Δ é um contexto simples sem declarações de variáveis ou *teorias*. Se $\Gamma(m)$ é uma declaração $m : \texttt{THEORY} = \Delta$, então $kind(\Gamma(m)) = \texttt{THEORY}$, e $definition(\Gamma(m)) = \Delta$.

Lembre-se de que na Definição 3.1.2 os contextos eram checados com respeito a um contexto vazio ($\tau()(\Gamma)$), e a partir de agora esta checagem nem sempre vai ser com respeito a um contexto vazio, ou seja, a definição de τ será modificada de forma que o *argumento contexto* não é sempre vazio.

Definição 3.4.1 [Operação Typechecking para Contextos com e sem *teorias*]:

$$\tau(\Theta)(\{\}) = \texttt{CONTEXT}$$

$$\tau(\Theta)(\Gamma, s : \texttt{TYPE} = T) = \texttt{CONTEXT}, \text{ se } \Gamma(s) \text{ e } \Theta(s) \text{ são indefinidos,}\ \tau(\Theta)(\Gamma) = \texttt{CONTEXT}, \text{ e}\ \tau(\Theta;\Gamma)(T) = \texttt{TYPE}$$

$$\tau(\Theta)(\Gamma, c : T) = \texttt{CONTEXT}, \text{ se } \Gamma(c) \text{ e } \Theta(c) \text{ são indefinidos,}\ \tau(\Theta)(\Gamma) = \texttt{CONTEXT}, \text{ e}\ \tau(\Theta;\Gamma)(T) = \texttt{TYPE}$$

$$\tau(\Theta)(\Gamma, x : \texttt{VAR}T) = \texttt{CONTEXT}, \text{ se } \Gamma(x) \text{ e } \Theta(x) \text{ são indefinidos,}\ \tau(\Theta)(\Gamma) = \texttt{CONTEXT}, \text{ e}\ \tau(\Theta;\Gamma)(T) = \texttt{TYPE}$$

$$\begin{array}{lcl}\tau(\Theta)(\Gamma, m : \texttt{THEORY} = \Delta) & = & \texttt{CONTEXT}, \text{ se } \Gamma(m) \text{ e } \Theta(m) \text{ são indefinidos,} \\ & & \Delta \text{ possui somente declarações de constantes e tipos,} \\ & & \tau(\Theta)(\Gamma) = \texttt{CONTEXT}, \text{ e} \\ & & \tau(\Theta;\Gamma)(\Delta) = \texttt{CONTEXT}\end{array}$$

Ao contrário de antes, agora tipos e constantes já não são mais apenas símbolos, mas podem ser compostos de nomes da forma $m.s$ onde m é o símbolo que determina o nome de uma *teoria* e s é o símbolo correspondente ao nome da constante ou do tipo. Em outras palavras, a partir de agora, qualquer referência ao nome da constante ou do tipo s, declarado em uma *teoria* m, fora desta *teoria* deve ser prefixado com o nome da *teoria*, isto é, $m.s$. Como é esperado modificações nas definições de δ e τ se fazem necessárias, enquanto que π e μ permanecem inalteradas.

Definição 3.4.2 [Tipo Expandido e Typechecking para Símbolos Prefixados]:

$$\begin{array}{lcl}\delta(\Gamma)(m.s) & = & \delta(\Gamma)(\eta(\Gamma, m)(definition(\Gamma(m)(s)))), \text{ se} \\ & & definition(\Gamma(m))(s) \text{ é não vazio.} \\ \delta(\Gamma)(m.s) & = & m.s \text{ se } definition(\Gamma(m))(s) \text{ é vazio.}\end{array}$$

$$\begin{array}{lcl}\tau(\Gamma)(m.s) & = & \texttt{TYPE}, \text{ se } kind(\Gamma(m)) = \texttt{THEORY} \text{ e} \\ & & kind(\Gamma(m)(s)) = \texttt{TYPE} \\ \tau(\Gamma)(m.s) & = & \delta(\Gamma)(\eta(\Gamma, m)(type(\Gamma(m)(s)))), \\ & & \text{se } kind(\Gamma(m)) = \texttt{THEORY} \text{ e} \\ & & kind(\Gamma(m)(s)) = \texttt{CONSTANT}\end{array}$$

onde $\eta(\Gamma, m)(a)$ é o resultado de prefixar cada símbolo de constante e tipo não prefixado em a por m, como um *indivíduo* ou uma expressão tipo.

O exemplo a seguir deixa estas novas idéias um pouco mais claras.

Exemplo 3.4.1: Seja Ω'' o contexto dado por:

$$\begin{array}{lcl}\Omega, \ \texttt{reals:THEORY} & = & (\texttt{real:TYPE}, \\ & & 0 : \texttt{real}, \\ & & \leq : [[\texttt{real}, \texttt{real}] -> \texttt{bool}], \\ & & \texttt{nonneg\_real} : \texttt{TYPE} = \{x : \texttt{real} \mid \ \leq (0, x)\}, \\ & & 1 : \texttt{nonneg\_real})\end{array}$$

Então:

$$\delta(\Omega'')(\texttt{reals.nonneg\_real}) = \{x : \texttt{reals.real} \mid \texttt{reals.} \leq (\texttt{reals.}0, x)\}$$

$$\tau(\Omega'')(\texttt{reals.nonneg\_real}) = \texttt{TYPE}$$

$$\tau(\Omega'')(\texttt{reals.}1 = \{x : \texttt{reals.real} \mid \texttt{reals.} \leq (\texttt{reals.}0, x)\}$$

3.4.2 *teorias* Parametrizadas

As *teorias* parametrizadas são obtidas permitindo que *teorias* possam ser declaradas como $m.[\Pi] : \texttt{THEORY} = \Delta$ onde Π é um contexto que contém uma *lista de parâmetros* e Δ é o *corpo da teoria*. Assim, se m ocorre em um contexto Γ, então Π é $formals(\Gamma(m))$, e Δ é $definition(\Gamma(m))$. No caso de *teorias* não parametrizadas $formals(\Gamma(m))$ é vazio. Correspondentemente as declarações de constantes e tipos são referenciadas, fora de suas respectivas *teorias* parametrizadas, como $m.[\sigma].s$ onde σ representa a lista dos parâmetros de que consiste os termos e tipos. Uma vez acrescentados os parâmetros a *operação typechecking* τ deve ser atualizada para que possamos fazer a checagem para os parâmetros formais esperados, assim como o operador δ, uma vez que os símbolos declarados na *teoria* são prefixados com seus nomes de *teorias* quando referenciados fora desta *teoria*.

Definição 3.4.3 [Operação Typechecking: *teorias* e Tipos Parametrizados]:

$$\tau(\Theta)(\Gamma, m[\Pi] : \texttt{THEORY} = \Delta) = \texttt{CONTEXT} \text{ se } \Gamma(m), \Theta(m) \text{ e } \Pi(m) \text{ são indefinidos}$$
$\tau(\Theta)(\Gamma) = \texttt{CONTEXT}$
$\tau(\Theta; \Gamma)(\Pi) = \texttt{CONTEXT}$
Π possui somente declarações de constantes e tipos, sem definições
$\tau(\Theta; \Gamma; \Pi)(\Delta) = \texttt{CONTEXT}$
Δ possui somente declarações de constantes e tipos

$$\tau(\Gamma)(m[\sigma].s) = \texttt{TYPE}, \text{ se}$$
$kind(\Gamma(m)) = \texttt{THEORY}$
$kind(\Gamma(m)(s)) = \texttt{TYPE}$ e
$\tau(\Gamma)(\Pi = \sigma) = \texttt{CONTEXT}$

$$\begin{aligned}
\tau(\Gamma)(m[\sigma].s) &= \delta(\Gamma)((\eta(\Gamma, m[\sigma])(type(\Gamma(m)(s))))), \\
&\quad \text{se } kind(\Gamma(m)) = \texttt{THEORY} \\
&\quad kind(\Gamma(m)(s)) = \texttt{CONSTANT} \text{ e} \\
&\quad \tau(\Gamma)(\Pi = \sigma) = \texttt{CONTEXT} \\
\\
\delta(\Gamma)(m[\sigma].s) &= \delta(\Gamma)((\eta(\Gamma, m[\sigma])(definition(\Gamma(m)(s))))), \text{ se} \\
&\quad definition(\Gamma(m)(s)) \text{ não é vazio} \\
\\
\delta(\Gamma)(m[\sigma].s) &= m[\sigma].s, \text{ se } definition(\Gamma(m)(s)) \text{ é vazio}
\end{aligned}$$

3.5 Os DataTypes em PVS

O PVS não permite definições recursivas de tipos, ou seja, uma declaração/definição de tipo em um contexto deve usar somente símbolos previamente declarados no contexto. Para contornar esta situação o PVS oferece um mecanismo chamado `DATATYPE`, o qual é uma forma de definição de tipo recursivo. Em outras palavras, o `DATATYPE` introduz um novo construtor de tipo que é a solução para uma equação recursiva da forma $T = \varphi[T]$ onde T é um tipo. Este mecanismo *abstract datatype* do PVS [50] foi, em parte, inspirado pelo *princípio shell* usado no Boyer-Moore [9], e diferentemente de outros sistemas, *Datatypes* são tipos primitivos em PVS.

Geralmente, a ocorrência do tipo T no lado direito da equação $T = \varphi[T]$ deve ser *positiva*, ou seja, deve satisfazer a seguinte definição:

Definição 3.5.1: Uma ocorrência do tipo `T` é positiva na expressão φ se, e somente se, umas das seguintes acontece:

(1) $\varphi \equiv$ `T`.

(2) `T` ocorre positivamente no supertipo φ' de φ.

(3) $\varphi \equiv [\varphi_1 \rightarrow \varphi_2]$ onde `T` ocorre positivamente em φ_2.

(4) $\varphi \equiv [\varphi_1, \cdots, \varphi_n]$ onde `T` ocorre positivamente em algum φ_i.

(5) $\varphi \equiv [\#l_1 : \varphi_1, \cdots, l_n : \varphi_n\#]$ onde `T` ocorre positivamente em algum φ_i.

(6) $\varphi \equiv datatype[\varphi_1, \cdots, \varphi_n]$, onde *datatype* é um `DATATYPE` previamente definido e `T` ocorre positivamente em φ_i, onde φ_i é um parâmetro positivo do *datatype*, ou seja, φ_i ocorre positivamente no tipo de um dos argumentos.

Por exemplo, `T` ocorre positivamente em `sequence[T]` onde `sequence[T]` é definida no prelude do PVS [52] como sendo o tipo funcional `[nat -> T]`.

A *operação typechecking* quando aplicada a uma declaração `DATATYPE` verifica o cumprimento de algumas regras as quais são a base para as declarações `DATATYPE`. As regras, que qualquer declaração `DATATYPE` deve obedecer, são as seguintes:

1. Os construtores devem possuir nomes distintos;
2. Os identificadores dos construtores devem possuir nomes distintos;
3. O identificador do construtor deve possuir nome diferente do nome do construtor;
4. O identificador usado para o `DATATYPE` não pode ser usado como um construtor;
5. O identificador usado para o `DATATYPE` não pode ser usado como um identificador de um construtor;
6. O identificador usado para o `DATATYPE` não pode ser usado como um argumento de um construtor;
7. Os argumentos dos construtores devem possuir nomes distintos;
8. Deve existir pelo menos um construtor não-recursivo, ou seja, um que não possui uma ocorrência recursiva do `DATATYPE` em seus argumentos.

Por exemplo, no prelude [52] podemos encontrar a seguinte especificação do *abstract datatypes* para listas de um dado tipo `T`.

Dessa forma, `list` é especificado como um tipo que é parametrizado com o tipo `T` e com dois construtores `null` e `cons`. O construtor `null` não toma argumentos e o construtor `cons` toma dois argumentos onde o primeiro, `car`, é de tipo `T` e o segundo, `cdr`, é um `list` o

```
list[T: TYPE] : DATATYPE
  BEGIN
    null: null?
    cons(car: T, cdr: list): cons?
  END
```

qual promove a chamada recursiva do `DATATYPE`. Os identificadores dos construtores `null` e `cons` são, respectivamente, `null?` e `cons?`, e o identificador usado para o `DATATYPE` é `list`.

Abaixo apresentamos um outro exemplo, um pouco mais complexo, de `DATATYPE`:

```
dt: DATATYPE
  BEGIN
   c0: c0?
   c1(a: int, b: {z: (even?) | z > a}, c: int): c1?
   c2(a: int, b: {n: nat | n > a}, c: int): c2?
  END dt
```

3.6 As Regras de Prova do PVS

A seguir apresentamos as regras de prova do PVS, as quais são apresentadas em termos de um cálculo de sequentes, cuja semântica é a semântica usual de Gentzen tal como apresentada na Seção 2.1.2, e cujas provas de correção são apresentadas em detalhe em [51]. A única diferença na notação dos sequentes apresentada a seguir e a apresentada na Seção 2.1.2 é que aqui levamos em conta o contexto, ou seja, o sequente é da forma $\Sigma \vdash_\Gamma \Lambda$, onde Γ é o contexto, Σ é o conjunto de fórmulas antecedentes, e Λ é o conjunto de fórmulas consequentes. Ademais, as regras de inferência são apresentadas no mesmo formato tal como mostra a Tabela 2.1.2.

Iniciamos apresentando as regras estruturais, as quais permitem rearranjar ou enfraquecer um sequente por meio da introdução de novas fórmulas sequentes dentro da conclusão. A seguir apresentamos uma regra de *enfraquecimento* (W), duas regras de *contração* ($C \vdash$) e ($\vdash C$) que permitem eliminar multíplas ocorrências de uma fórmula, duas regras de *comutação* ($X \vdash$) e ($\vdash X$) as quais nos dizem que a ordem das fórmulas

tanto no antecedente quanto no consequente é irrelevante.

$$\frac{\Sigma_1 \vdash_\Gamma \Lambda_1}{\Sigma_2 \vdash_\Gamma \Lambda_2}\ (W) \qquad \text{se } \Sigma_1 \subseteq \Sigma_2 \text{ e } \Lambda_1 \subseteq \Lambda_2$$

$$\frac{a, \Sigma \vdash_\Gamma \Lambda}{a, a, \Sigma \vdash_\Gamma \Lambda}\ (C \vdash) \qquad \frac{\Sigma \vdash_\Gamma a, \Lambda}{\Sigma \vdash_\Gamma a, a, \Lambda}\ (\vdash C)$$

$$\frac{\Sigma_1, b, a, \Sigma_2 \vdash_\Gamma \Lambda}{\Sigma_1, a, b, \Sigma_2 \vdash_\Gamma \Lambda}\ (X \vdash) \qquad \frac{\Sigma \vdash_\Gamma \Lambda_1, b, a, \Lambda_2}{\Sigma \vdash_\Gamma \Lambda_1, a, b, \Lambda_2}\ (\vdash X)$$

A *regra do corte*, chamada de (Cut), também é usada no PVS para introduzir o caso *split* sobre uma fórmula a dentro de uma prova do sequente $\Sigma \vdash_\Gamma \Lambda$, ou seja, pode ser usada para fornecer dois novos objetivos a saber, $\Sigma, a \vdash_\Gamma \Lambda$ e $\Sigma \vdash_\Gamma a, \Lambda$, os quais podem ser vistos como assumindo a em um dos ramos de prova e $\neg a$ no outro.

$$\frac{(\tau(\Gamma)(a) \sim \texttt{bool})_\Gamma \quad \Sigma, a \vdash_\Gamma \Lambda \quad \Sigma \vdash_\Gamma a, \Lambda}{\Sigma \vdash_\Gamma \Lambda}\ (Cut)$$

Nas regras para *axiomas proposicionais* temos a regra (AX) cujo significado é trivial, e a regra ($\texttt{FALSE} \vdash$) e a regra ($\vdash$ $\texttt{TRUE}$) nos dizem que uma ocorrência de `FALSE` no antecendente ou uma ocorrência de `TRUE` no consequente nos leva a uma prova trivial, ou seja, um axioma.

$$\frac{}{\Sigma, a \vdash_\Gamma a, \Lambda}\ (AX)$$

$$\frac{}{\Sigma, \texttt{FALSE} \vdash_\Gamma \Lambda}\ (\texttt{FALSE} \vdash) \qquad \frac{}{\Sigma \vdash_\Gamma \texttt{TRUE}, \Lambda}\ (\vdash \texttt{TRUE})$$

Quando uma fórmula e/ou uma definição de constante já estão no contexto, então elas valem trivialmente. Isto é o que nos dizem as *regras de contexto* $(ContextFormula)$ e $(ContextDefinition)$ abaixo. Por outro lado, podemos estender um contexto com fórmulas antecedentes ou negações de fórmulas consequentes, assim como, enfraquecer um contexto. Como mostram, respectivamente, as regras $(Context \vdash)$, $(\vdash Context)$ e $(ContextW)$.

$$\frac{}{\vdash_\Gamma a}\ (ContextFormula) \qquad \text{se } a \text{ é uma fórmula em } \Gamma$$

$$\frac{}{\vdash_\Gamma s = a}\ (ContextDefinition) \qquad \text{se } s : T = a \text{ é uma definição de constante em } \Gamma$$

$$\frac{\Sigma, a \vdash_{\Gamma,a} \Lambda}{\Sigma, a \vdash_\Gamma \Lambda}\ (Context \vdash) \qquad \frac{\Sigma \vdash_{\Gamma,\neg a} \Lambda}{\Sigma \vdash_\Gamma a, \Lambda}\ (\vdash Context)$$

$$\frac{\Sigma \vdash_\Gamma \Lambda}{\Sigma \vdash_{\Gamma'} \Lambda}\ (ContextW) \qquad \text{se } \Gamma \text{ é um prefixo de } \Gamma'$$

A *regras condicionais* a seguir, indicam como eliminar um `IF-THEN-ELSE` em uma prova.

$$\frac{\Sigma, a, b \vdash_{\Gamma,a} \Lambda \quad \Sigma, c \vdash_{\Gamma,\neg a} a, \Delta}{\Sigma, \mathtt{IF}(a, b, c) \vdash_\Gamma \Lambda}\ (\mathtt{IF} \vdash)$$

$$\frac{\Sigma, a \vdash_{\Gamma,a} b, \Lambda \quad \Sigma \vdash_{\Gamma,\neg a} a, c, \Delta}{\Sigma \vdash_\Gamma \Lambda, \mathtt{IF}(a, b, c)}\ (\vdash \mathtt{IF})$$

Nas regras abaixo a notação $a[e]$ é usada para destacar uma ou mais ocorrências de e na fórmula a tal que não existem ocorrências de variáveis livres em e. Da mesma forma a notação $\Lambda[e]$ serve para destacar as ocorrências de e em Λ. Estas regras são as chamadas *regras de igualdade*. Observe que as regras para transitividade e simetria podem ser obtidas de $(Refl)$ e $(Repl)$.

$$\frac{}{\Sigma \vdash_\Gamma a = a, \Lambda}\ (Refl) \qquad \frac{a = b, \Sigma[a] \vdash_\Gamma \Lambda[a]}{a = b, \Sigma[b] \vdash_\Gamma \Lambda[b]}\ (Repl)$$

Nas *regras de igualdade booleana*, a regra $(Refl\ \mathtt{TRUE})$ nos diz que uma fórmula antecedente a pode ser tratada como uma igualdade antecedente da forma $a = \mathtt{TRUE}$, e similarmente, a regra $(Repl\ \mathtt{FALSE})$ nos diz que uma fórmula consequente a pode ser tratada como uma igualdade antecedente da forma $a = \mathtt{FALSE}$. Por outro lado, a regra $(\mathtt{TRUE} - \mathtt{FALSE})$ nos diz que as contantes booleanas `TRUE` e `FALSE` são distintas.

$$\frac{\Sigma[\mathtt{TRUE}], a \vdash_\Gamma \Lambda[\mathtt{TRUE}]}{\Sigma[a], a \vdash_\Gamma \Lambda[a]}\ (Refl\ \mathtt{TRUE}) \qquad \frac{\Sigma[a] \vdash_\Gamma a, \Lambda[a]}{\Sigma[\mathtt{FALSE}], a \vdash_\Gamma \Lambda[\mathtt{FALSE}]}\ (Repl\ \mathtt{FALSE})$$

$$\frac{}{\Sigma, \texttt{TRUE} = \texttt{FALSE} \vdash_\Gamma \Lambda}\ (\texttt{TRUE} - \texttt{FALSE})$$

Como *regras de redução* temos a β-redução e a projeção de produtos.

$$\frac{}{\vdash_\Gamma (\lambda(x:T):a)(b) = a[b/x]}\ (\beta) \qquad \frac{}{\vdash_\Gamma p_i(a_1, a_2) = a_i}\ ,\ i = 1, 2 \quad (\pi)$$

Regras de extensionalidade são para determinar se duas funções ou dois produtos são iguais. Estas verificações são feitas, respectivamente, verificando se duas função f e g fornecem resultados iguais quando aplicadas a um argumento arbitrário, e o produto verificando se as projeções correspondentes são iguais.

$$\frac{\Sigma \vdash_{\Gamma, s:A} (f\ s) =_{B[s/x]} (g\ s), \Lambda}{\Sigma \vdash_\Gamma f =_{x:A \to B} g, \Lambda}\ ,\ se\ \Gamma(s)\ é\ indefinido \quad (FunExt)$$

$$\frac{\Sigma \vdash_\Gamma p_1(a) =_{T_1} p_1(b), \Lambda \qquad \Sigma \vdash_\Gamma p_2(a) =_{T_2[(p_1\ a)/x]} p_2(b), \Lambda}{\Sigma \vdash_\Gamma a =_{[x:T_1 T_2]} b, \Lambda}\ (TupExt)$$

Finalmente a *regra de restrição de tipo* vem suprir a necessidade de uma regra que introduza a restrição de tipo como uma fórmula antecedente em um sequente.

$$\frac{\tau(\Gamma)(a) = A \quad \pi(A)(a), \Sigma \vdash_\Gamma \Lambda}{\Sigma \vdash_\Gamma \Lambda}\ (Typepred)$$

Capítulo 4

Formalização da Teoria de Sistemas Abstratos de Redução

Neste capítulo apresentamos uma *teoria* em PVS para Sistemas Abstratos de Redução, chamada `ars`, a qual é uma versão estendida do trabalho apresentado previamente em [19]. Como veremos nas seções seguintes, a utilização de uma linguagem de especificação de ordem superior, como PVS, permite expressar (formalizar) de maneira natural propriedades de objetos de segunda-ordem como as *relações de redução*. Assim, as formalizações dos conceitos que apresentamos no decorrer deste texto muito se aproximam das formulações analíticas tradicionais contidas na Teoria de Reescrita em geral, permitindo expressar, aplicar e visualizar conceitos e propriedades envolvendo relações de redução de uma forma quasi-geométrica, isto é, por meio de *diagramas*, como usual em ARS.

Além das vantagens mencionadas, uma outra característica importante da semântica operacional do PVS é permitir ao usuário um bom controle da estrutura das provas durante o seu desenvolvimento. Este controle é obtido através do uso de *estratégias de prova* [40,66] que são uma combinação de *regras de prova*, as quais modificam uma árvore de prova por meio de aplicações de *regras lógicas* (Veja Seção 3.6).

Dentre vários aspectos interessantes da formalização da teoria de ARS apresentada neste trabalho, como será visto, destacamos a formalização do Princípio de Indução Noeteriana tal como mostrado na Seção 4.5. A correção deste princípio de indução foi formalizada com base no princípio conhecido como Princípio de Indução Bem-Fundada. Dessa forma, é possível formalizar teoremas da teoria de ARS como o bem conhecido

Lema de Newman. Em particular, apresentamos na Seção 4.6 as formalizações deste lema e do Lema de Yokohuchi, as quais foram previamente publicadas em [21].

Adotamos como metodologia neste capítulo apresentar as definições, conceitos e resultados, nos padrões de ARS e em seguida apresentar sua especificação em PVS. Ressaltamos, que não apresentamos as provas feitas em PVS, mas somente as especificações de definições e teoremas.

4.1 Estrutura Hierárquica da *Teoria* `ars`

A *teoria* `ars` foi desenvolvida parametrizada (Ver Seção 3.4) com um tipo `T` "fixo" não interpretado, ou seja, `ars[T]`, e é composta por uma hierarquia de *sub-teorias*, também parametrizadas com o mesmo tipo `T`, nas quais especificamos noções básicas, e formalizamos propriedades e resultados sobre ARS e que servem de base para formalizar muitas outras propriedades, além daquelas já formalizadas em `ars`.

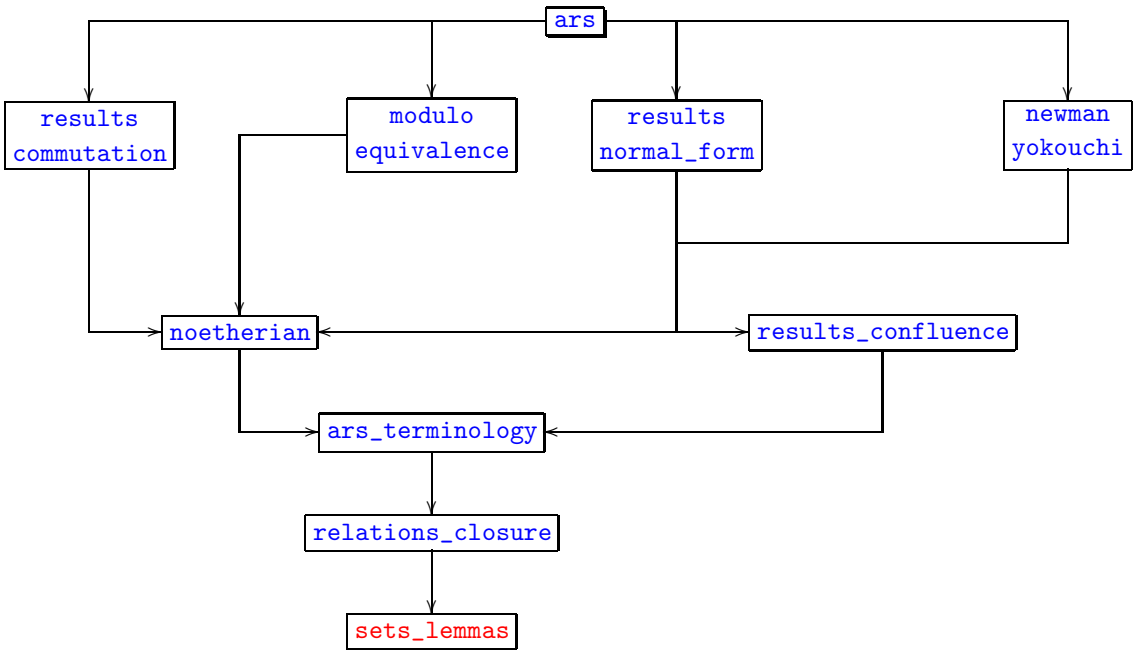


Figura 4.1.1: Estrutura Hierárquica da *Teoria* `ars`

Na Figura 4.1.1 apresentamos a estrutura hierárquica da *teoria* `ars` e nas seções seguintes apresentamos, parcialmente, o que foi especificado e formalizado em cada *sub-teoria*

de `ars`.

Na Figura 4.1.1, a *sub-teoria* `sets_lemmas` fornece lemas sobre propriedades da Teoria dos Conjuntos e faz parte da biblioteca do PVS e pode ser encontrada no *prelude* [52]. Um *conjunto* em PVS é um tipo polimórfico, chamado `set`, e é especificado na teoria `sets` do *prelude* como um predicado, isto é,

```
set:TYPE = setof[T]
```

onde

```
setof:TYPE = [T -> bool].
```

Além do tipo `set`, na teoria `sets` podemos encontrar especificações sobre as operações usuais da Teoria dos Conjuntos, como por exemplo, as apresentadas na Tabela 4.1.1 onde `A` e `B` são do tipo `set`, e `IUnion(B)` é um operador de tipo `set` que faz a união de uma família `B` de subconjuntos de `T`.

Tabela 4.1.1: Operações da Teoria dos Conjuntos

```
member(x, A): bool = A(x)

subset?(A, B): bool = (FORALL x: member(x, A) => member(x, B))

union(A, B): set = {x | member(x, A) OR member(x, B)}

IUnion(B): set[T] = {x | EXISTS i: B(i)(x)}
```

4.2 Fechos de uma Relação

Um ARS é definido como sendo um par $(A, \rightarrow)$, onde $\rightarrow$ é uma relação binária sobre o conjunto A, i.e., $\rightarrow \subseteq A \times A$, como mostra a Definição 4.2.1.

Da mesma forma que em ARS, toda a *teoria* `ars` é baseada sobre relações binárias. Portanto, iniciamos o desenvolvimento da *teoria* `ars` criando a *sub-teoria* `relations_closure`

(Ver Figura 4.1.1) a qual contém definições e propriedades relacionadas com fecho de uma relação binária [56].

Definição 4.2.1 [Relação Binária]: Sejam A e B dois conjuntos e seja $A \times B$ o produto cartesiano de A por B, ou seja, o par ordenado $(a, b) \in A \times B$ se, e somente se, $a \in A$ e $b \in B$. Qualquer subconjunto $\rightarrow$ de $A \times B$ é denominado uma relação binária de A em B. Em particular, se $\rightarrow$ é uma relação de A em A, isto é, se $\rightarrow$ é um subconjunto de $A \times A$, dizemos simplesmente que $\rightarrow$ é uma relação binária sobre A. Se $(a, b) \in \rightarrow$, então escrevemos $a \rightarrow b$.

Em PVS, uma relação binária `R` é tratada como um predicado sobre um tipo não interpretado `T`, ou seja, a relação `R` possui o tipo `PRED[[T, T]]` onde

```
PRED: TYPE = [[T,T] -> bool].
```

Observe que em acordo com a Definição 4.2.1 o conjunto A é tratado como o tipo não interpretado `T`, e a relação $\rightarrow$ é tratada como a relação `R`. Além disso, `R(x,y)` significa que `x` *reduz para* `y`. Neste sentido, entendemos por *redução* a transformação passo a passo de algum objeto (por exemplo, um termo). A concretização da idéia de redução é possível através do conceito *composição de relações binárias* definido a seguir:

Definição 4.2.2 [Composição de Relações]: Se $\rightarrow_1$ é uma relação binária de A em B e $\rightarrow_2$ uma relação binária de B em C, então a composição de $\rightarrow_1$ e $\rightarrow_2$, denotada por $\rightarrow_1 \circ \rightarrow_2$, é dada por:

$$\rightarrow_1 \circ \rightarrow_2 = \{(x, z) \mid \exists y \in B : x \rightarrow_1 y \; e \; y \rightarrow_2 z\}$$

É fácil observar que a composição de relações binárias é associativa, ou seja,

$$\rightarrow_1 \circ(\rightarrow_2 \circ \rightarrow_3) = (\rightarrow_1 \circ \rightarrow_2)\circ \rightarrow_3$$

Com base nesta definição podemos definir a i-ésima iteração de uma relação binária $\rightarrow$ sobre A, denotada por $\rightarrow^i$ e dada recursivamente como segue:

$$\rightarrow^i = \begin{cases} \{(x, x) : x \in A\} & \text{se} \quad i = 0 \\ \rightarrow \circ \rightarrow^{i-1} & \text{se} \quad i > 0 \end{cases}$$

Estas definições já vem pré-construídas no *prelude* e na biblioteca do PVS e são apresentadas na Tabela 4.2.1, onde o operador `iterate(R,i)`, para todo `i` ≥ 0, especifica a relação $\underbrace{\texttt{R} \circ \texttt{R} \cdots \circ \texttt{R}}_{\texttt{i vezes}}$, e o operador `=[T]` especifica a *relação identidade* $\rightarrow^0 := \{(x, x) \mid x \in A\}$.

Tabela 4.2.1: Composição de Relações Binárias

```
(R1 o R2)(x, z): bool = EXISTS y: R1(x, y) AND R2(y, z)

iterate(R, i): RECURSIVE PRED[[T, T]] =
               IF i = 0 THEN =[T] ELSE iterate(R, i - 1) o R ENDIF
               MEASURE i
```

Uma das vantagens oferecidas pelo construtor `iterate(R, i)` é permitir obter provas indutivas sobre o comprimento `i`. Em particular, em se tratando da *teoria* `ars` este construtor permite obter provas indutivas sobre o comprimento das reduções, como veremos na Seção 4.4.

Definição 4.2.3 [Relações Especiais]: (a) Uma relação binária $\rightarrow$ sobre A é:

1. *reflexiva* se $\forall x \in A : x \rightarrow x$;
2. *simétrica* se $\forall x, y \in A : x \rightarrow y \ \Rightarrow \ y \rightarrow x$;
3. *transitiva* se $\forall x, y, z \in A : x \rightarrow y \ e \ y \rightarrow z \ \Rightarrow \ x \rightarrow z$;
4. uma *relação de equivalência* se $\rightarrow$ é reflexiva, simétrica e transitiva.

(b) Definimos a *relação inversa* $\rightarrow^{-1}$ sobre A, também denotada por $\leftarrow$, como sendo

$$\rightarrow^{-1} = \{(y, x) \in A \times A \mid (x, y) \in \rightarrow\}.$$

As relações especiais dadas na Definição 4.2.3 são especificadas como os predicados `reflexive?`, `symmetric?`, `transitive?` e `equivalence?` os quais juntamente com o operador `converse`, que representa a relação inversa, são encontrados no *prelude* do PVS e apresentados na Tabela 4.2.2.

Definição 4.2.4 [Fecho de uma Relação]: Sejam $\rightarrow$ uma relação e P um conjunto de propriedades. Então, o fecho de $\rightarrow$ com relação a P é a menor relação que contém $\rightarrow$ e

Tabela 4.2.2: Relações Especiais

```
reflexive?(R): bool = FORALL x: R(x, x)

symmetric?(R): bool = FORALL x, y: R(x, y) IMPLIES R(y, x)

transitive?(R): bool = FORALL x, y, z: R(x, y) AND R(y, z) IMPLIES R(x, z)

equivalence?(R): bool = reflexive?(R) AND symmetric?(R) AND transitive?(R)

converse(R): PRED[[T, T]] = (LAMBDA (y: T), (x: T): R(x, y))
```

que satisfaz as propriedades em P. Em particular, se $\rightarrow$ é uma relação binária sobre A e $P = \{$reflexiva$\}$ ou $\{$simétrica$\}$ ou $\{$transitiva$\}$ ou $\{$reflexiva, transitiva$\}$ ou $\{$reflexiva, transitiva, simétrica$\}$ temos, respectivamente, os fechos reflexivo, simétrico, transitivo, reflexivo e transitivo, e de equivalência, respectivamente, denotados por $\rightarrow^{=}$, $\leftrightarrow$, $\rightarrow^{+}$, $\rightarrow^{*}$ e $\leftrightarrow^{*}$, e dados como segue:

Definição Abstrata	`Especificação em PVS`
$\rightarrow^{=} := \rightarrow \cup \rightarrow^{0}$	fecho reflexivo (`RC`)
$\leftrightarrow := \rightarrow \cup \leftarrow$	fecho simétrico (`SC`)
$\rightarrow^{+} := \bigcup_{i>0} \rightarrow^{i}$	fecho transitivo (`TC`)
$\rightarrow^{*} := \rightarrow^{+} \cup \rightarrow^{0}$	fecho reflexivo e transitivo (`RTC`)
$\leftrightarrow^{*} := (\leftrightarrow)^{*}$	fecho de equivalência (`EC`)

Assim, dizemos que:

1. $a \rightarrow^{n} b$ se existe um "caminho" de comprimento n de a para b.

2. $a \rightarrow^{*} b$ se existe algum "caminho" de comprimento ≥ 0 (finito) de a para b.

3. $a \rightarrow^{+} b$ se existe algum "caminho" de comprimento > 0 (finito) de a para b.

Como citamos antes, estes conceitos são especificados em PVS na *sub-teoria* `relations_closure`. Nossa especificação dos fechos de uma relação, mostrada na tabela 4.2.3, foi desenvolvida a partir do desenvolvido por Alfons Geser na *teoria* `closure_ops` que pode ser encontrada em [58]. Apenas mudamos os nomes das definições e provamos algumas propriedades adicionais.

Tabela 4.2.3: Teoria `relations_closure.pvs`

```
relations_closure[T : TYPE] : THEORY
BEGIN

 IMPORTING orders@closure_ops[T], sets_lemmas[T]

   S, R: VAR pred[[T, T]]
   n: VAR nat
   p: VAR posnat

   reflexive: TYPE = (reflexive?)
   symmetric: TYPE = (symmetric?)
   transitive: TYPE = (transitive?)

   reflexive_transitive?(R): bool = reflexive?(R) AND transitive?(R)

   reflexive_transitive: TYPE = (reflexive_transitive?)
   equivalence: TYPE = (equivalence?)

    RC(R): reflexive = union(R, =)
...
    SC(R): symmetric = union(R, converse(R))
...
    TC(R): transitive = IUnion(LAMBDA p: iterate(R, p))
...
   RTC(R): reflexive_transitive = IUnion(LAMBDA n: iterate(R, n))
...
    EC(R): equivalence = RTC(SC(R))
...
END relations_closure
```

Na tabela 4.2.3 os tipos `(reflexive?)`, `(symmetric?)`, `(transitive?)` e `(equivalence?)`, são gerados, respectivamente, pelos predicados apresentados na Tabela 4.2.2. Os três pontos ... que aparecem na Tabela 4.2.3, e no decorrer de todo este texto, significam especificações e/ou formalizações de conceitos e/ou resultados omitidas.

4.3 Terminologia Básica

Na *sub-teoria* `ars_terminology` (Veja Figura 4.1.1), especificamos as noções mais importantes de ARS as quais formam uma base sólida para o desenvolvimento de toda a *teoria* `ars`, ou seja, a partir desta *sub-teoria* foi, e é possível especificar vários conceitos e

formalizar vários resultados da teoria de ARS, como veremos na próxima seção.

No decorrer de todo este trabalho consideramos um ARS $(A, \rightarrow)$ arbitrário. Também, durante todo este texto consideraremos, a menos que se diga o contrário, $x, y, z \in A$, ou equivalentemente, `x,y,z:T`, e consideramos `=>`, `<=>` e `&` como abreviações para `IMPLIES`, `IFF` e `AND`, respectivamente. Além disso, como é usual na literatura de reescrita, nas Figuras 4.3.1, 4.3.2 e 4.3.3, e daqui por diante, as setas tracejadas significam *existência*.

Definição 4.3.1: (a) Dizemos que:

- x é *redutível* se, e somente se, existe um y tal que $x \rightarrow y$.
- x está na *forma normal* (*irredutível*) se, e somente se, x não é redutível.
- y é uma forma normal de x se, e somente se, $x \rightarrow^* y$ e y está em forma normal.
- y é um *sucessor direto* de x se, e somente se, $x \rightarrow y$.
- y é um *sucessor* de x se, e somente se, $x \rightarrow^+ y$.
- x e y são *juntáveis* se, e somente se, existe um z tal que $x \rightarrow^* z \ ^*\!\leftarrow y$, neste caso denotamos por $x \downarrow y$.

(b) Uma relação $\rightarrow$ é chamada (ver Figuras 4.3.1, 4.3.2 e 4.3.3)

- **Confluente** se, e somente se, $\forall x, y, z: \ y \ ^*\!\leftarrow x \rightarrow^* z \ \Rightarrow \ y \downarrow z$
- **semi-confluente** se, e somente se, $\forall x, y, z: \ y \leftarrow x \rightarrow^* z \ \Rightarrow \ y \downarrow z$
- **Church-Rosser** se, e somente se, $\forall x, y: \ x \leftrightarrow^* y \ \Rightarrow \ x \downarrow y$
- **localmente confluente** se, e somente se, $\forall x, y, z: \ y \leftarrow x \rightarrow z \ \Rightarrow \ y \downarrow z$
- **fortemente confluente** se, e somente se, $\forall x, y, z: \ y \leftarrow x \rightarrow z \ \Rightarrow \ \exists w.\ y \rightarrow^* w \ ^=\!\leftarrow z$
- **terminante ou noetheriana** se, e somente se, não existem cadeias de redução infinita $x_0 \rightarrow x_1 \rightarrow \cdots$
- **fracamente terminante** se, e somente se, todo elemento possui uma forma normal.
- **convergente** se, e somente se, $\rightarrow$ é confluente e terminante.

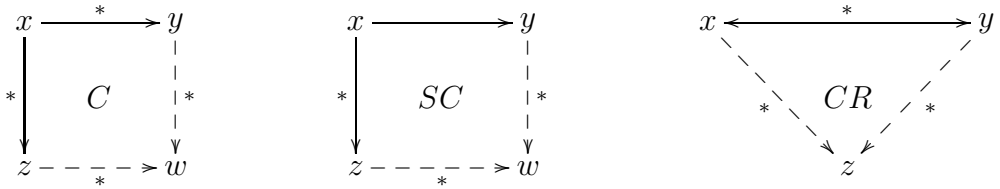


Figura 4.3.1: Confluente, Semi-Confluente e Propriedade Church-Rosser

(c) Uma relação $\rightarrow$ possui a **propriedade do diamante** se, e somente se,

$$\forall x, y, z : \ y \leftarrow x \rightarrow z \ \Rightarrow \ \exists w. \ y \rightarrow w \leftarrow z$$

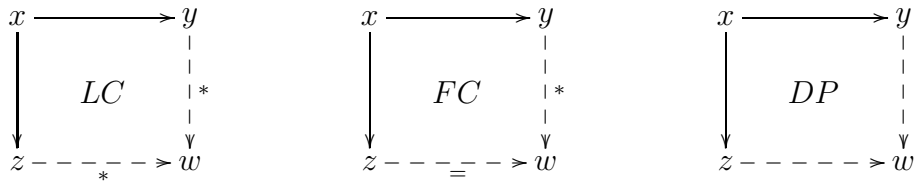


Figura 4.3.2: Local Confluente, Fortemente Confluente e Propriedade do Diamante

(d) Duas relações $\rightarrow_1$ e $\rightarrow_2$

- **comutam** se, e somente se, $\forall x, y, z : \ y \ {}^{*}_{1}\!\leftarrow x \rightarrow^{*}_{2} z \ \Rightarrow \ \exists w. \ y \rightarrow^{*}_{2} w \ {}^{*}_{1}\!\leftarrow z$
- **comutam fortemente** se, e somente se, $\forall x, y, z : \ y \ {}_{1}\!\leftarrow x \rightarrow_{2} z \ \Rightarrow \ \exists w. \ y \rightarrow^{=}_{2} w \ {}^{*}_{1}\!\leftarrow z$
- possui a **propriedade do diamante comutativa** se, e somente se,

$$\forall x, y, z : \ y \ {}_{1}\!\leftarrow x \rightarrow_{2} z \ \Rightarrow \ \exists w. \ y \rightarrow_{2} w \ {}_{1}\!\leftarrow z$$

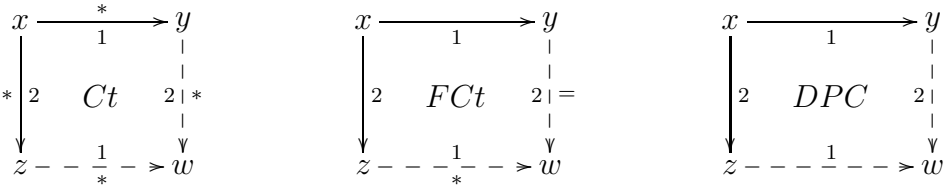


Figura 4.3.3: Comutam, Comutam Fortemente e Propriedade do Diamante Comutativa

A Tabela 4.3.1 apresenta, em parte, a terminologia apresentada na Definição 4.3.1. Como podemos observar as especificações de tais conceitos são apresentadas de forma bastante transparente e extremamente próximas das formulações analíticas. Isto, sem dúvida nenhuma, permite uma maior transparência na especificação de outros conceitos e formalização de resultados da teoria de ARS, como veremos nas próximas seções.

Tabela 4.3.1: Teoria ars_terminology.pvs

```
ars_terminology[T : TYPE] : THEORY
BEGIN

  IMPORTING relations_closure[T]

   R, R1, R2 : VAR PRED[[T, T]]
  x, y, z, r : VAR T

 reducible?(R)(x): bool = EXISTS y: R(x,y)
...
 is_normal_form?(R)(x): bool = NOT reducible?(R)(x)

 normal_form?(R)(x,y): bool = RTC(R)(x,y) & is_normal_form?(R)(y)

 normalizing?(R): bool = FORALL x: EXISTS y: normal_form?(R)(x,y)

 has_unique_nf?(R,x): bool = EXISTS y: normal_form?(R)(x,y)
                                 &  FORALL z: normal_form?(R)(x,z)  => y = z

 unique_nf?(R)(x,y): bool = normal_form?(R)(x,y)
                                 &  FORALL z: normal_form?(R)(x,z)  => y = z

 joinable?(R)(x,y): bool = EXISTS z: RTC(R)(x,z) & RTC(R)(y, z)

 church_rosser?(R): bool = FORALL x, y: EC(R)(x,y) => joinable?(R)(x,y)

 local_confluent?(R): bool = FORALL x, y, z: R(x,y) & R(x,z) =>
                                                      joinable?(R)(y,z)

 semi_confluent?(R): bool = FORALL x, y, z: R(x,y) & RTC(R)(x,z) =>
                                                     joinable?(R)(y,z)

 confluent?(R): bool = FORALL x, y, z: RTC(R)(x,y) & RTC(R)(x,z) =>
                                                     joinable?(R)(y,z)

 strong_confluent?(R): bool = FORALL x, y, z: R(x,y) & R(x,z) =>
                                           EXISTS r: RTC(R)(y,r) & RC(R)(z,r)

 diamond_property?(R): bool = FORALL x, y, z: R(x,y) & R(x,z) =>
                                                  EXISTS r: R(y,r) & R(z,r)

 commute?(R1,R2): bool = FORALL x, y, z: RTC(R1)(x,y) & RTC(R2)(x,z) =>
                                         EXISTS r: RTC(R2)(y,r) & RTC(R1)(z,r)

 strong_commute?(R1,R2): bool = FORALL x, y, z: R1(x,y) & R2(x,z) =>
                                           EXISTS r: RC(R2)(y,r) & RTC(R1)(z,r)

 locally_commute?(R1,R2): bool = FORALL x, y, z: R1(x,y) & R2(x,z) =>
                                           EXISTS r: RTC(R2)(y,r) & RTC(R1)(z,r)
...
END ars_terminology
```

Além disso, como veremos na sequência, já se tornou padrão associar conceitos da Teoria de Reescrita à *diagramas*, por exemplo, confluência e comutação, como mostrado na Definição 4.3.1. Esta associação permite um tratamento num estilo quase-geométrico dos resultados e suas provas, facilitando o entendimento e a explicação.

4.4 Confluência, Comutação e Formas Normais

As *sub-teorias* `results_confluence`, `results_commutation` e `results_normal_form` (Veja Figura 4.1.1) foram desenvolvidas com a intenção de confirmar que as especificações dadas na Tabela 4.3.1 realmente formam uma base sólida para o desenvolvimento de resultados da teoria de ARS. Sem dúvida, o nosso propósito foi alcançado e nesta seção apresentamos as formalizações de alguns resultados básicos envolvendo confluência, a propriedade Church-Rosser, comutação e outros, tais como, o Lema da União Comutativa e o Lema da Comutação.

Teorema 4.4.1: (a) Se $\rightarrow$ é confluente então $\rightarrow$ é semi-cofluente.

(b) Se $\rightarrow$ é semi-cofluente então $\rightarrow$ possui a propriedade Church-Rosser.

(c) $\rightarrow$ possui a propriedade Church-Rosser se, e somente se, $\rightarrow$ é confluente.

Demonstração:

(a) Obviamente, por definição, qualquer relação confluente é semi-confluente.

(b) Se $\rightarrow$ é semi-confluente e $x \leftrightarrow^* y$ então devemos mostrar que $x \downarrow y$, ou seja, $\rightarrow$ possui a propriedade Church-Rosser. A demonstração se dá por indução no comprimento da cadeia de redução $x \leftrightarrow^n y$. Se $x = y$, é trivial. Agora suponha que

$$x \leftrightarrow^* y \leftrightarrow z$$

Pela hipótese de indução existe w tal que $x \rightarrow^* w \ {}^*\!\leftarrow y$. Nos resta, então mostrar que $x \downarrow z$. Para isto distinguimos dois casos (veja diagramas na Figura 4.4.1):

$y \leftarrow z$: Neste caso $x \downarrow z$ segue diretamente de $x \downarrow y$.

$y \rightarrow z$**:** Finalmente, a semi-confluência implica $w \downarrow z$ e consequentemente $x \downarrow z$.

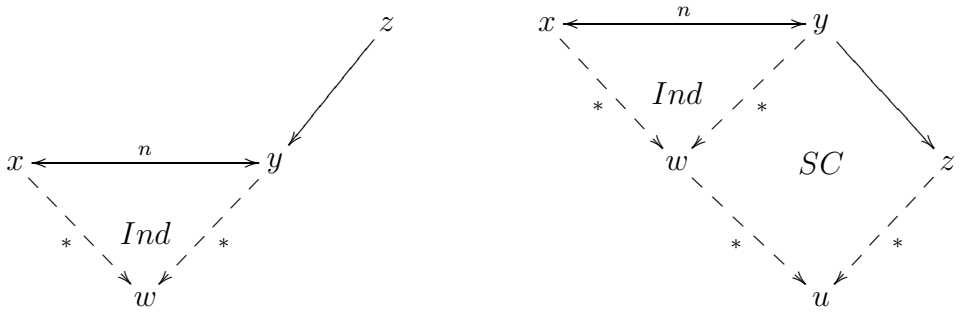


Figura 4.4.1: Semi-confluência implica a propriedade Church-Rosser

(c) Se $\rightarrow$ possui a propriedade Church-Rosser e $y \,{}^{*}\!\leftarrow x \rightarrow^{*} z$ então $y \leftrightarrow^{*} z$ e portanto, $y \downarrow z$, ou seja, $\rightarrow$ é confluente. A outra direção segue imediatamente pelos itens anteriores. ■

O Teorema 4.4.1 foi formalizado na *sub-teoria* `results_confluence` e sua especificação é apresentada na Tabela 4.4.1. Para formalizar os resultados apresentados na Tabela 4.4.1 usamos não mais do que as técnicas apresentadas na demostração anterior, ou seja, o lema `Confl_implies_Semi` foi provado simplesmente expandindo as definições de `confluent?` e `semi_confluent?`, e instanciando adequadamente as variáveis quantificadas decorrentes destas expansões.

Tabela 4.4.1: Especificação do Teorema 4.4.1

```
Confl_implies_Semi: THEOREM confluent?(R) => semi_confluent?(R)

Semi_implies_CR: THEOREM semi_confluent?(R) => church_rosser?(R)

CR_iff_Confluent: THEOREM church_rosser?(R) <=> confluent?(R)
```

A prova do resultado `Semi_implies_CR` segue na íntegra os casos e diagramas tal como apresentado na Figura 4.4.1. No entanto, formalizamos a parte referente à indução sobre n como um lema auxiliar, tal como mostrado na Tabela 4.4.2.

Para provar o lema `semi_and_iterate` invocamos o *comando de prova* `(induct n)` pré-construído no PVS. Este comando aplica no sequente corrente o esquema de indução

Tabela 4.4.2: Formalização da parte indutiva do item b) do Teorema 4.4.1.

```
semi_and_iterate: LEMMA FORALL (n: nat): semi_confluent?(R) &
                                         iterate(SC(R), n)(x,y)
                                          =>
                                            joinable?(R)(x,y)
```

natural dividindo-o em dois outros sequentes os quais correspondem à base de indução (`n=0`) e ao passo de indução. Note que o esquema de indução é proporcionado pelo operador `iterate(SC(R), n)`. Um outro *comando de prova* que usamos na prova do lema `semi_and_iterate`, e em várias outras, é o comando `(prop)`. Este comando nos permite distinguir os casos $y \leftarrow z$ e $y \rightarrow z$ mencionados no teorema acima. Em outras palavras, o comando `(prop)` gera dois sequentes onde em um deles o objetivo é provar o caso $y \leftarrow z$ e no outro é provar o caso $y \rightarrow z$. De maneira geral, o comando `(prop)` aplica simplificações proposicionais, principalmente eliminado conectivos proposicionais, e corresponde à regra `IF-THEN-ELSE` apresentada na Seção 3.6. Para mais detalhe sobre estes *comandos de prova* ver Apêndice C e [66]. Note que os comandos de prova citados acima proporcionam um bom controle da estrutura das provas.

O resultado `CR_iff_Confluent` segue, também na íntegra, a demonstração sugerida no item (c). Isto é: por um lado, a prova segue expandindo as definições `confluent?` e `church_rosser?`, instanciando convenientemente as variáveis quantificadas, e manipulando os fechos da relação `R`. Por outro lado, aplicando diretamente os lemas `Confl_implies_Semi` e `Semi_implies_CR`.

Ressaltamos que a maneira natural com que foram especificados os conceitos básicos da teoria ARS, apresentados na Tabela 4.3.1, nos permitiu visualizar as provas mecânicas dos resultados, apresentados na Tabela 4.4.1, de uma maneira muito mais clara, assim como, outros resultados formalizados na *sub-teoria* `results_confluence`, dos quais destacamos `Str_Confl_implies_Semi_Confl`, `Strong_Confl_implies_Confl` e `DP_implies_StC`, apresentados na Tabela 4.4.3.

Outros exemplos de formalizações, que foram baseados em diagramas, e que vieram a reforçar o quanto as especificações apresentadas na Tabela 4.3.1 são coerentes para

os nossos objetivos, podem ser encontrados na *sub-teoria* `results_commutation` (Ver Figura 4.1.1) onde formalizamos, dentre outros, o *Lema da Comutação* e o *Lema da União Comutativa*, cujas especificações são apresentadas na Tabela 4.4.3.

Tabela 4.4.3: Outros resultados básicos.

```
Str_Confl_implies_Semi_Confl: THEOREM strong_confluent?(R) => semi_confluent?(R)

Strong_Confl_implies_Confl: COROLLARY strong_confluent?(R) => confluent?(R)

DP_implies_StC: LEMMA diamond_property?(R) => strong_confluent?(R)

Commutative_Union_Lemma: THEOREM confluent?(R1) & confluent?(R2) &
                                 commute?(R1,R2) => confluent?(union(R1, R2))

Comutation_Lemma: THEOREM strong_commute?(R1,R2) => commute?(R1,R2)
```

Para verificar o quão adequadas são as especificações envolvendo formas normais, dentre outros, formalizamos na *sub-teoria* `results_normal_form` (Ver Figura 4.1.1) o seguinte resultado:

Lema 4.4.2: Se $\rightarrow$ é confluente e fracamente terminante, então todo elemento possui uma única forma normal.

Também, formalizamos na *sub-teoria* `results_normal_form` um dos principais resultados de ARS, seguindo os mesmos padrões mencionados anteriormente. A saber,

Teorema 4.4.3: Se $\rightarrow$ é confluente e fracamente terminante, então $x \leftrightarrow^* y$ se, e somente se, $x \downarrow = y \downarrow$. Onde $x \downarrow$ significa que x possui uma única forma normal.

4.5 Formalização do Princípio de Indução Noeteriana

Na *sub-teoria* `noetherian` (Veja Figura 4.1.1) especificamos a noção de relação noeteriana e o formalizamos o Princípio de Indução Noeteriana, o qual é a base para as formalizações do Lema de Newman e do Lema de Yokouchi, como veremos na próxima seção. No entanto, antes de apresentarmos estas formalizações recordemos do Princípio de *Indução*

Bem-Fundada:

$$\forall x \in X.\ ((\forall y \in X.\ y < x\ \Rightarrow\ P(y))\ \Rightarrow\ P(x))\ \Longrightarrow\ \forall x \in X.\ P(x)$$

onde X é um conjunto qualquer, P é alguma propriedade sobre os elementos de X e $<$ é uma relação bem-fundada sobre X [69]. Em outras palavras, para provar que a propriedade $P(x)$ vale para todo $x \in X$ é suficiente mostrar que $P(x)$ vale, sabendo que $P(y)$ vale para todo $y < x$ (hipótese de indução).

Formalmente, o *Princípio de Indução Noeteriana* é expresso como segue:

$$(\forall x \in A.\ (\forall y \in A.\ x \rightarrow^{+} y\ \Rightarrow\ P(y))\ \Rightarrow\ P(x))\ \Longrightarrow\ \forall x \in A.\ P(x)$$

onde P é alguma propriedade sobre os elementos de A. Em outras palavras, para provar que $P(x)$ vale para todo x, é suficiente provar que $P(x)$ vale, sabendo que $P(y)$ vale para todos os sucessores y de x (hipótese de indução). O Princípio de Indução Noeteriana é uma generalização do Princípio de Indução Bem-Fundada sobre um conjunto bem-ordenado $(X, <)$ para sistemas de redução terminante (noeteriano) $(A, \rightarrow)$.

Como podemos ver em [4], é possível provar que:

Teorema 4.5.1: $\rightarrow$ é terminante (noeteriana) se, e somente se, o Princípio de Indução Noeteriana vale.

Demonstração: ($\Rightarrow$): Assuma que o Princípio de Indução Noeteriana (**PIN**) não vale para $\rightarrow$, ou seja, existe algum P tal que a premissa de **PIN** vale mas a conclusão não vale, ou seja, $\neg P(a_0)$ para algum $a_0 \in A$. Mas então a premissa de **PIN** implica que deve existe algum a_1 tal que $a_0 \rightarrow^{+} a_1$ e $\neg P(a_1)$. Pelo mesmo argumento, deve existir algum a_2 tal que $a_1 \rightarrow^{+} a_2$ e $\neg P(a_2)$. Seguindo com este raciocínio, concluímos que existe uma cadeia infinita $a_0 \rightarrow^{+} a_1 \rightarrow^{+} a_2 \rightarrow^{+} \cdots$, ou seja, $\rightarrow$ não é terminante o que nos leva a uma contradição.

($\Leftarrow$): Considere o **PIN** onde $P(x)$:=*não existe cadeia infinita iniciando x.* O passo indutivo é: se não existe uma cadeia infinita iniciando em qualquer sucessor de x, então não existe cadeia infinita iniciando x. Portanto, a premissa de **PIN** vale e podemos concluir que $P(x)$ vale para todo x, ou seja, $\rightarrow$ é terminante. ∎

Observe que o Princípio de Indução Bem-Fundada trabalha com antecessores e o Princípio de Indução Noeteriana com sucessores. Dessa forma, podemos caracterizar uma relação terminante (noeteriana) como segue:

Definição 4.5.2: $\rightarrow$ é terminante (noeteriana) se, e somente se, $\leftarrow$ é bem-fundada.

Tabela 4.5.1: Teoria `noetherian.pvs`

```
noetherian[T: TYPE] : THEORY
BEGIN

  IMPORTING ars_terminology[T], orders@well_foundedness[T]

     P: VAR PRED[T]
     R: VAR PRED[[T, T]]
  x, y: VAR T

 noetherian?(R): bool = well_founded?(converse(R))

 noetherian: TYPE = (noetherian?)
...
 noetherian_induction: LEMMA
             (FORALL (R: noetherian, P: PRED[T]):
               (FORALL x:
                  (FORALL y: TC(R)(x, y) => P(y)) => P(x))
            =>
              (FORALL x: P(x)))
END noetherian
```

Como mencionamos no início deste capítulo, o princípio de indução noeteriana é formalizado usando o Princípio de Indução Bem-Fundada, o qual foi formalizado em PVS e pode ser encontrado no *prelude* (lema `wf_induction`), assim como a noção de relação bem-fundada. Agora, as especificações da noção de relação noeteriana e do Princípio de Indução Noeteriana são apresentadas na Tabela 4.5.1. Observe que o Princípio de Indução Noeteriana exige a quantificação de uma relação, ou seja, é um objeto de ordem superior, e que a linguagem de ordem superior do PVS nos permitiu especificar este princípio de uma forma natural e elegante.

4.6 Formalizações dos Lemas de Newman e Yokouchi

Na *sub-teoria* `newman_yokouchi` (Veja Figura 4.1.1) está a formalização do Lemma de Newman [47] e do Lema de Yokouchi [71] previamente publicadas em [21]. Como mencionado em outro momento, o nosso objetivo com estas formalizações não é apresentar mais uma formalização de resultados não elementares da teoria ARS, mas evidenciar que a *teoria* `ars` efetivamente é uma teoria de aspecto geral suficiente para formalizar vários resultados da teoria de ARS. Ambas as formalizações utilizam o Princípio de Indução Noetheriana, apresentado na seção precedente, instanciado convenientemente de acordo com as exigências de cada lema.

4.6.1 Lema de Newman

Apesar de existir, segundo Bognar [8], mais de 50 maneiras de demonstrar o Lema de Newman: *sob a hipótese de terminação, confluência é equivalente a confluência local*[1], por exemplo, usando *Multisets* [6], por *Decreasing Diagrams* [8], e é claro a própria prova de Newman [47], em nossa formalização apresentamos a clássica prova por *aplicação do Princípio de Indução Noeteriana* dada por Huet em [34], a qual é um clássico exemplo de prova em lógica de ordem superior e é geralmente apresentada, por padrão, na literatura de reescrita [4].

Ressaltamos que a formalização do Lema de Newman, desenvolvida neste trabalho, se difere um pouco da formalização desenvolvida por Ruiz-Reina et al em [61]. Isto porque em [61] os autores formalizam o Lema de Newman inspirados pela prova dada por Klop em [42], a qual consiste em mostrar que a *relação de redução* possui a propriedade Church-Rosser em vez de lidar com confluência. No entanto, note que estes dois conceitos são equivalentes como mostrado na Tabela 4.4.1.

Lema 4.6.1 [Lema Newman [47]]: Seja R uma relação noeteriana definida sobre o conjunto A. Então R é confluente se, e somente se, R é localmente confluente.

[1]Em geral, confluência é indecidível. No entanto, é decidível para sistemas de reescrita finitos e terminantes.

Demonstração: A direção confluente implica localmente confluente, segue por definição. Na outra direção, suponha que R é noeteriana e localmente confluente. A confluência é provada usando o princípio de indução noeteriana com o predicado

$$P(x) = \forall b, c.\ b\ {}^*\!\leftarrow x \rightarrow^* c \implies b \text{ e } c \textit{ juntáveis}$$

Obviamente, R é confluente se $P(x)$ vale para todo x. De acordo com o princípio de indução noeteriana devemos provar que $P(x)$ vale supondo que $P(t)$ vale para todo t tal que $x \rightarrow^+ t$. Para provar $P(x)$, analisamos a divergência $b\ {}^*\!\leftarrow x \rightarrow^* c$. Se $x = b$ ou $x = c$, temos claramente que b e c são juntáveis. Caso contrário, temos que $x \rightarrow x_1 \rightarrow^* b$ e $x \rightarrow x_2 \rightarrow^* c$ como mostra o diagrama na Figura 4.6.1.

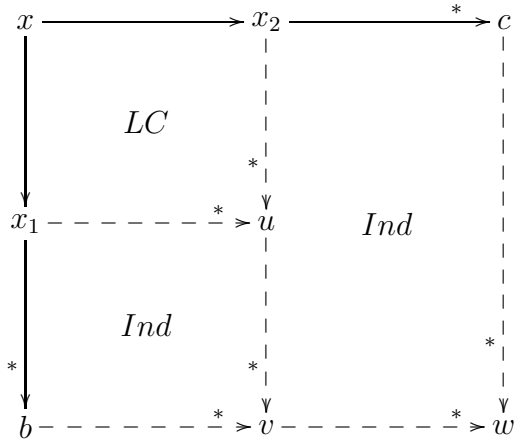


Figura 4.6.1: Prova do Lema de Newman.

A existência de u segue por local confluência (LC) de R, a existência de v e w segue pela hipótese de indução (Ind) pois $x \rightarrow^+ x_1$ e $x \rightarrow^+ x_2$, respectivamente. ■

A especificação do Lema de Newman é apresentada na Tabela 4.6.1. A sua prova mecânica foi completada após a aplicação de 143 comandos (regras) de prova, e segue na íntegra a prova (analítica) apresentada acima, isto é: a direção ($\Rightarrow$) segue expandindo as definições de `confluent?(R)` e `local_confluent?(R)`, e instanciando adequadamente as variáveis quantificadas decorrentes destas expansões; e a direção ($\Leftarrow$) é provada invocando e instanciando adequadamente o lema `noetherian_induction` com o predicado

```
LAMBDA (a: T): (FORALL (b, c: T):
                  RTC(R)(a, b) AND RTC(R)(a, c) IMPLIES joinable?(R)(b,c))
```

Tabela 4.6.1: Sub-teoria `newman_yokouchi`: Lema de Newman

```
newman_yokouchi[T : TYPE] : THEORY
BEGIN
  IMPORTING results_confluence[T], noetherian[T]
  R, S: VAR PRED[[T, T]]

 Newman_lemma: THEOREM
   noetherian?(R) => (confluent?(R) <=> local_confluent?(R))
...
END newman_yokouchi
```

Assim, o principal objetivo a ser provado, `confluent?(R)`, segue trivialmente assumindo que a hipótese a seguir vale

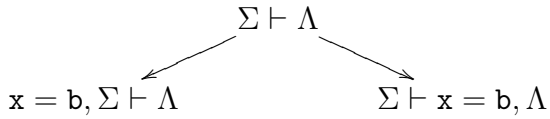
```
FORALL (x: T):
     FORALL (b, c: T): RTC(R)(x, b) AND RTC(R)(x, c) IMPLIES joinable?(R)(b, c)
```

Consequentemente um novo sequente é gerado onde o principal objetivo é provar que a hipótese acima vale, supondo que a condição a seguir vale

```
FORALL (y: T):
        TC(R)(x, y) IMPLIES
         (FORALL (b, c: T):
            RTC(R)(y, b) AND RTC(R)(y, c) IMPLIES joinable?(R)(b, c))
```

Este novo objetivo é provado seguindo na íntegra os casos e o diagrama apresentado na Figura 4.6.1.

Dentre os comandos usados para provar este lema destacamos os seguintes: o comando `(split)`, o qual dentre outras ações, divide um sequente que possui uma fórmula consequente da forma $A \wedge B$ em dois sequentes nos quais os objetivos são as fórmulas consequentes A e B, foi usado para dividir a prova de `<=>` em dois sub-objetivos, ou seja, `<=` e `=>`; o comando `(case)` foi usado para dividir a prova de `<=` em casos como sugerido na prova acima. Por exemplo, ao aplicarmos a regra de prova `case`, digamos `(case x = b)`, a um sequente $\Sigma \vdash \Lambda$, dois sub-objetivos (sequentes) são gerados:

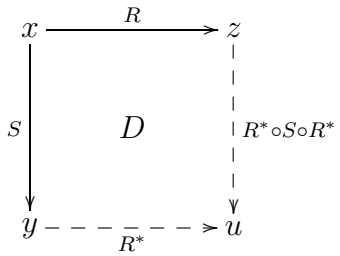


De outra forma, a regra de prova `(case x = b)` divide o sequente $\Sigma \vdash \Lambda$ em dois outros sequentes, onde em um deles `x = b` aparece no antecedente, ou seja, é assumido como verdade, e no outro `x = b` aparece no consequente, ou seja, como uma obrigação de prova. Observe que a regra `case` é equivalente à regra do corte (olhada de "baixo para cima") apresentada na Seção 3.6; e o comando `(typepred)` o qual corresponde à regra (*Typepred*) apresentada na Seção 3.6, ou seja, é usado para introduzir explicitamente, no antecedente de um sequente, o predicado que define o tipo de um elemento. Por exemplo, se `i` é do tipo `nat` então o comando `(typepred "i")` introduz no antecedente de um sequente o predicado que define o tipo de `i`, isto é, `i>=0`.

4.6.2 Lema de Yokouchi

Na sequência apresentamos a formalização do Lema de Yokouchi. Este lema também foi formalizado em Coq por Säibi [63] e foi a chave para formalizar a confluência do Cálculo de Substituições Explícitas $\lambda_{\sigma\Uparrow}$. Além disso, este lema é mais um exemplo de como a associação com diagramas facilita o entendimento e a formalização de conceitos não-triviais.

Lema 4.6.2 [Lema de Yokouchi [71]]: Sejam R e S duas relações definidas sobre o mesmo conjunto T, com R confluente e noeteriana, e S tendo a propriedade do diamante. Suponha ainda que o seguinte diagrama vale:



Então a relação $R^* \circ S \circ R^*$ possui a propriedade do diamante.

Demonstração: A prova se inicia com a generalização D' do diagrama D, como mostrado na Figura 4.6.2. Esta generalização é provada por indução noeteriana usando o predicado

$$P(x) := \forall y, z.\ xR^*z\ \wedge\ xSy\ \Rightarrow\ \exists u.(yR^*u\ \wedge\ zR^* \circ S \circ R^*u)$$

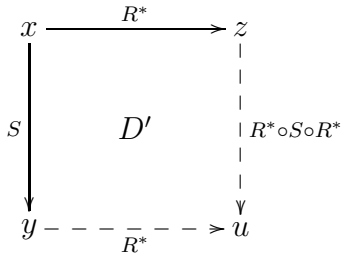


Figura 4.6.2: Generalização do Diagrama D como D'

Então, para provar que $R^* \circ S \circ R^*$ possui a propriedade do diamante usamos novamente o princípio de indução noeteriana com o seguinte predicado

$$P'(x) := \forall y, z.\ xR^* \circ S \circ R^* y \ \wedge\ xR^* \circ S \circ R^* z \ \Rightarrow\ \exists u.(yR^* \circ S \circ R^* u \ \wedge\ zR^* \circ S \circ R^* u)$$

concluímos a demonstração, por indução no comprimento da redução do primeiro R^* em $xR^* \circ S \circ R^* y$. Em outras palavras, analisamos doi casos: $xR \circ R^* \circ S \circ R^* y$ e $xS \circ R^* y$ como mostram os diagramas apresentados, respectivamente, nas Figuras 4.6.3 e 4.6.4, onde C e DP representam a confluência de R e a propriedade do diamante de S, respectivamente dados como hipóteses.

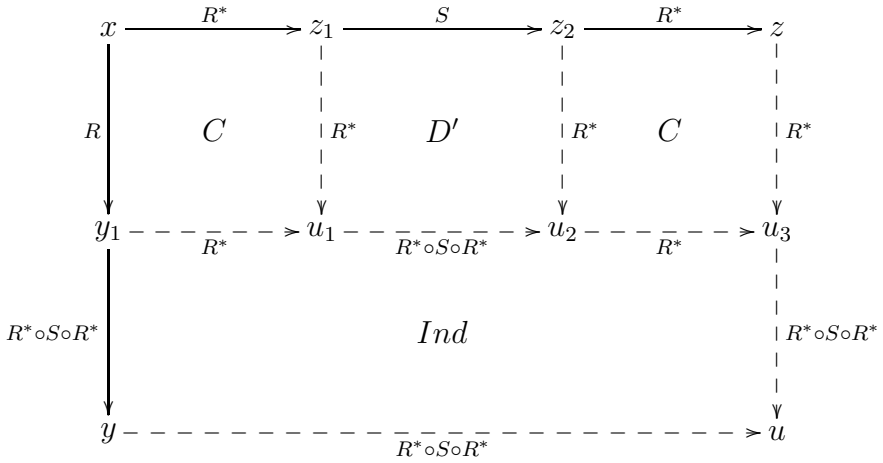


Figura 4.6.3: Caso $xR \circ R^* \circ S \circ R^* y$

■

A formalização do Lema de Yokouchi foi dividida em dois lemas, cujas especificações são apresentadas na Tabela 4.6.2. O primeiro lema, `Yokouchi_lemma_ax1`, apresentado na

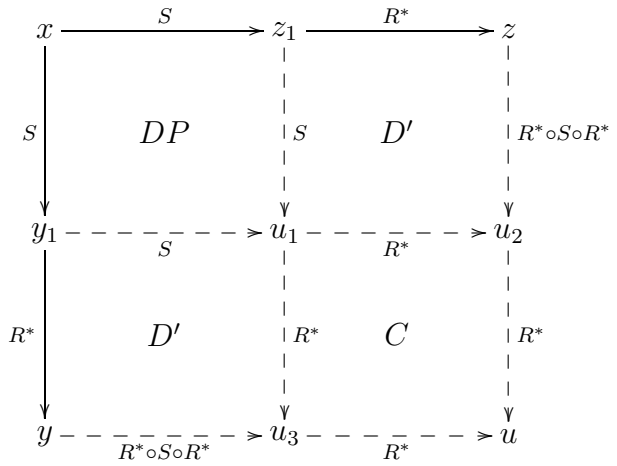


Figura 4.6.4: Caso $xS \circ R^*y$

Tabela 4.6.2, corresponde à formalização da generalização D' do diagrama D apresentado na Figura 4.6.2, e o segundo lema corresponde ao Lema 4.6.2.

Tabela 4.6.2: Sub-teoria `newman_yokouchi`: Lema de Yokouchi

```
newman_yokouchi[T : TYPE] : THEORY
BEGIN
  IMPORTING results_confluence[T], noetherian[T]
  R, S: VAR PRED[[T, T]]
...
 Yokouchi_lemma_ax1: LEMMA
  (noetherian?(R) & confluent?(R) &
   (FORALL x,y,z: (S(x,y) & R(x,z)) =>
    (EXISTS (u:T): RTC(R)(y,u) & (RTC(R) o S o RTC(R))(z,u))))
    =>     (FORALL x,y,z: (S(x,y) & RTC(R)(x,z)) =>
              (EXISTS (w:T): RTC(R)(y,w) & (RTC(R) o S o RTC(R))(z,w)))

 Yokouchi_lemma: THEOREM
  (noetherian?(R) & confluent?(R) & diamond_property?(S) &
   (FORALL x,y,z: (S(x,y) & R(x,z)) =>
    (EXISTS (u:T): RTC(R)(y,u) & (RTC(R) o S o RTC(R))(z,u))))
    =>     diamond_property?(RTC(R) o S o RTC(R))
END newman_yokouchi
```

A prova mecânica do lema `Yokouchi_lemma_ax1`, assim como do lema `Yokouchi_lemma` seguem na íntegra os passos apresentados na prova (analítica) acima. A formalização do lema `Yokouchi_lemma` foi completada após a aplicação de 266 comandos (regras) de prova, sem contar os comandos aplicados para provar o lema `Yokouchi_lemma_ax1`, e exigiu duas

aplicações do Princípio de Indução Noeteriana, ou seja, duas invocações do lema `noetherian_induction` apresentado na Tabela 4.5.1. De maneira geral, o lema `Yokouchi_lemma` é provado seguindo os diagramas das Figuras 4.6.3 e 4.6.4 de acordo com os seguintes passos:

1. Primeiro passo: introduz as constantes de Skolem e considera os casos `x` $= z_1$ e/ou `x` $= y_1$.

2. Segundo passo: invoca o lema `iterate_RTC` o qual estabelece que para todo `n`, `iterate(R,n)` $\subseteq$ `RTC(R)`; expande as definições de composição de relações, `confluent?` e outras; e aplica simplificação disjuntiva.

3. Terceiro passo: aplica a confluência de `R`, o lema auxiliar `Yokouchi_lemma_ax1` e a hipótese de indução para concluir.

Capítulo 5

Formalização da Teoria de Sistemas de Reescrita de Termos

Neste capítulo apresentamos a formalização de um conjunto de conceitos que formam a base da teoria de TRS. Essa formalização compõe uma *teoria* em PVS chamada `trs`, a ser disseminada em [20]. Assim como a *teoria* `ars`, o principal objetivo do desenvolvimento da *teoria* `trs` é fornecer uma formalização de conceitos básicos que possibilite o desenvolvimento de vários outros conceitos ou resultados da teoria de TRS. O conceito de *termo*, ou o *conjunto dos termos bem-formados*, foi especificado como um tipo recursivo com base em dois tipos não interpretados não vazios para símbolos de variável e símbolos de função associadas a sua aridade. Dessa forma, os *termos* bem-formados serão variáveis ou aplicações de símbolos de função a uma seqüência de termos de comprimento correspondente à aridade do símbolo de função. Com base nessa formalização de *termos* é possível especificar indutivamente as noções de *posições* e *subtermos*. Pode-se então formular noções mais complexas como *substituições* e sua extensão homeomorfa para termos e noções elaboradas como a de *par crítico* de maneira natural, tal como apresentado na literatura de reescrita. Seguindo esse caminho obtemos uma formalização robusta para a teoria de TRS.

Como evidência da generalidade e robustez da *teoria* `trs` apresenta-se uma formalização, dentre outros resultados, do conhecido Teorema dos Pares Críticos de Knuth-Bendix. Como mencionado na introdução, até onde sabemos, esta é a primeira formalização completa deste teorema numa linguagem de especificação de ordem superior como o PVS.

Em TRS, como já mencionamos antes, usa-se um conjunto pré-definido de regras de redução para se fazer computações. Por exemplo, no λ-*cálculo*, reduções são executadas através de β-*redução*, dentre outras. Assim, fazendo um paralelo com ARS observa-se que um "conjunto" de regras de redução define uma ou mais relações de redução (binárias) as quais são construídas sobre o *termos*. Dessa forma, todos os conceitos e resultados sobre ARS, vistos no capítulo anterior, podem ser naturalmente importados para TRS. E o fato de a *teoria* `ars` ser parametrizada com um tipo não interpretado `T`, isto é, `ars[T]`, nos permitiu importar todos os seus conceitos e resultados para a *teoria* `trs`, simplesmente instânciando o parâmetro `T` com o tipo `term` (Ver Seção 5.2).

5.1 Estrutura Hierárquica da Teoria `trs`

Com a mesma metodologia adotada no capítulo anterior, na Figura 5.1.1 apresentamos a estrutura hierárquica da *teoria* `trs` na qual aparece destacada em azul a *teoria* `ars`. Na seqüência apresentamos, parcialmente, o que foi formalizado em cada *sub-teoria*.

Na Figura 5.1.1 as *sub-teorias* `identity` e `finite_sequences` já vem pré-construídas no PVS, e podem ser encontradas no *prelude* [52]. A *sub-teoria* `finite_sequences` define *seqüências finitas* como um tipo dependente (Ver Seção 3.3) dado por:

```
finite_sequence:  TYPE = [# length:nat, seq:[below[length] -> T] #].
```

onde `below` é um subconjunto, possivelmente vazio, dos números naturais, também definido como um tipo dependente, e que pode ser encontrado no *prelude*, e é dado por:

```
below(i:nat):  TYPE = {s:nat | s < i}.
```

Além disso, na *sub-teoria* `finite_sequences`, a concatenação de duas seqüências finitas `fs1` e `fs2` é definida através do operador denotado por "`o`" e apresentado na Tabela 5.1.1.

Como veremos no decorrer deste capítulo, muitos dos elementos da *teoria* `trs` são baseados em seqüências finitas, por exemplo termos e posições, e várias formalizações de

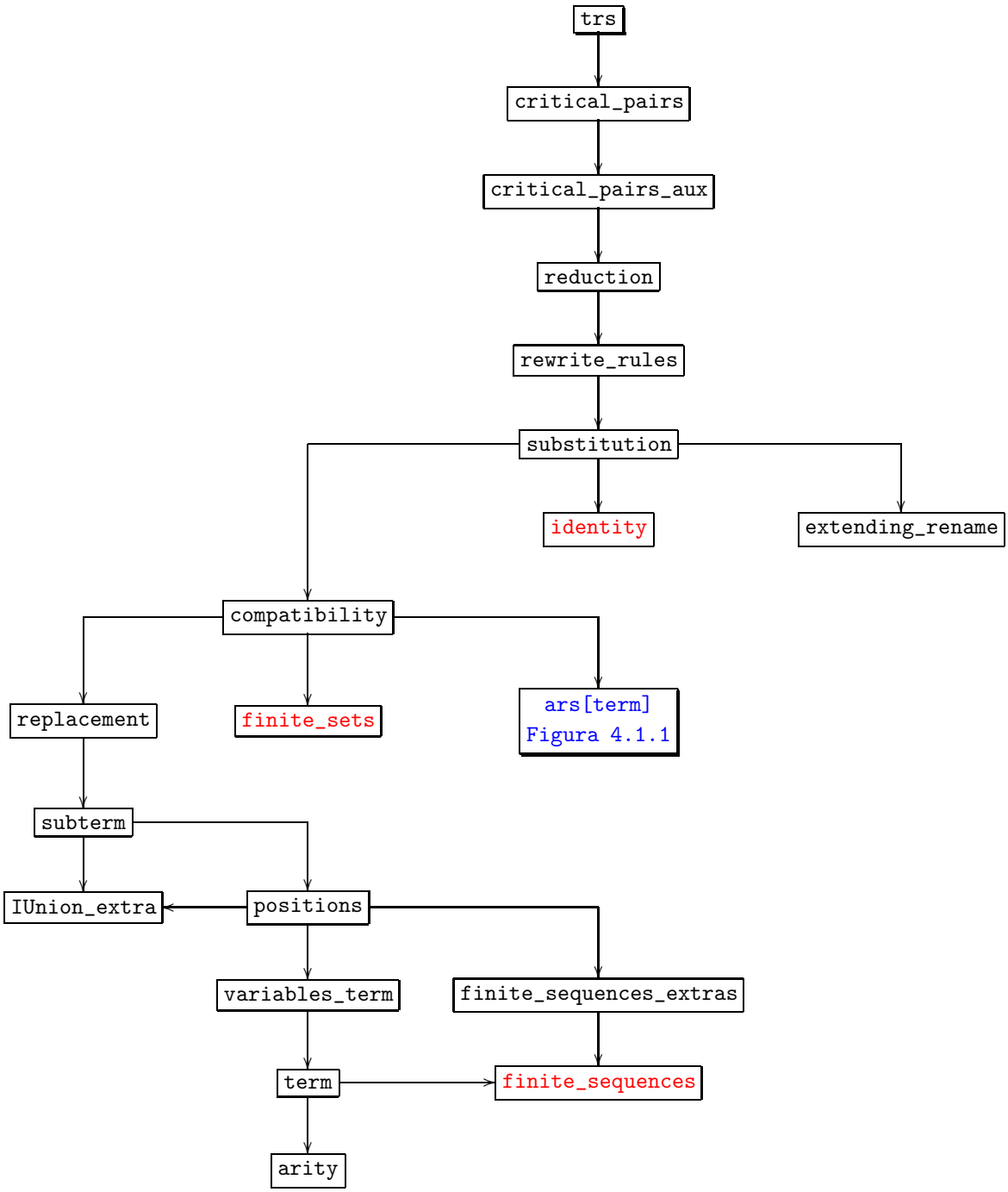


Figura 5.1.1: Estrutura Hierárquica da Teoria `trs`

resultados da *teoria* `trs` são obtidas por indução no comprimento de uma seqüência finita. Além disso, o manuseio de seqüências finitas é essencial na especificação de subtermos, substituições, entre outros.

Devido ao supracitado, complementamos a *sub-teoria* `finite_sequences` desenvolvendo a *sub-teoria* `finite_sequences_extras` na qual formalizamos várias propriedades

Tabela 5.1.1: Concatenação de duas seqüências finitas.

```
(fs1 o fs2): finite_sequence =
   LET   l1 = fs1‘length,
       lsum = l1 + fs2‘length IN
       (# length := lsum,
            seq := (LAMBDA (n:below[lsum]):
                   IF n < l1
                      THEN fs1‘seq(n)
                      ELSE fs2‘seq(n-l1)
                   ENDIF) #);
```

inerentes à Teoria de Sequências Finitas. Por exemplo, especificamos a idéia de inserir (`insert?`) um elemento `x` numa posição qualquer de uma seqüência finita `seq`, e em seguida, especificamos o construtor `add_first` o qual insere um elemento na primeira posição de uma seqüência finita `seq` tal como mostrado na Tabela 5.1.2. Também na Tabela 5.1.2 apresentamos um construtor, `replace`, para trocar o elemento na posição `n` de uma seqüência finita `seq` por um elemento `x`.

Tabela 5.1.2: Alguns Construtores Definidos na Teoria `finite_sequences_extras`

```
insert?(x, seq, (n: upto[length(seq)])): finseq =
        (# length := seq‘length + 1,
             seq := (LAMBDA (i: below[seq‘length + 1]):
                         (IF i < n THEN seq(i)
                          ELSIF i = n THEN x
                          ELSE seq(i - 1) ENDIF)) #)

add_first(x, seq): finseq = insert?(x, seq, 0)

replace(x, seq, (n: below[length(seq)])): finseq =
      (IF seq‘length = 0 THEN seq
       ELSE
        (# length := seq‘length,
             seq := (LAMBDA (i: below[seq‘length]):
                         (IF i < n THEN seq(i)
                          ELSIF i = n THEN x
                          ELSE seq(i) ENDIF)) #)
       ENDIF)
```

Na Tabela 5.1.2, `seq(i)` representa o i-ésimo elemento da seqüência finita `seq`, e `upto` é um subconjunto dos números naturais definido como um tipo dependente no *prelude* e

dado por

```
upto(i:nat):  NONEMPTY_TYPE = {s:nat | s <= i} CONTAINING i.
```

A *sub-teoria* `identity` (Veja Figura 5.1.1) simplesmente define a *função identidade* como uma função bijetiva sobre um tipo `T` e é dada por:

```
identity:  (bijective?[T,T]) = (LAMBDA x:  x).
```

Em adição acrescentamos o conjunto dos números naturais não-nulos `upto?` tal como apresentados na Tabela 5.1.3. Assim como a *sub-teoria* `finite_sequence_extra` a *sub-teoria* `IUnion_extra` (Veja Figura 5.1.1) foi desenvolvida como uma teoria parametrizada, cujo parâmetro é um tipo não interpretado `T`. Esta *sub-teoria*, `IUnion_extra`, contém formalizações de alguns resultados auxiliares para lidar com a união finita de subconjuntos de um dado tipo `T`. Em particular, estamos interessados na união finita de imagens finitas de uma dada função, tal como apresentado na Tabela 5.1.3.

Tabela 5.1.3: A união finita de conjuntos finitos é finito

```
IUnion_extra[T: TYPE]: THEORY
  BEGIN
     i, n: VAR nat
...
     upto?(n: posnat): NONEMPTY_TYPE = {i: posnat | i <= n} CONTAINING n
...
     IUnion_of_finite_is_finite: LEMMA
       FORALL (n: posnat, (f:[upto?(n) -> set[T]])):
         (FORALL (i: upto?(n)): is_finite(f(i)))
          => is_finite(IUnion(LAMBDA (i: upto?(n)): f(i)))
...
END IUnion_extra
```

5.2 Termos

Um TRS é definido sobre um conjunto de objetos chamados *termos*, e na *sub-teoria* `term` (Veja Figura 5.1.1) especificamos a estrutura destes *termos* de acordo com a Definição

5.2.1, e mantendo a mesma visão de generalidade que tivemos para `ars`, desenvolvemos a *teoria* `trs` como uma teoria parametriza, dada por:

```
trs[variable:TYPE+, symbol:TYPE+]
```

onde `TYPE+` significa um tipo não vazio, o tipo `variable` representa um conjunto V arbitrário de *símbolos de variável*, e o tipo `symbol` representa um conjunto Σ de *símbolos de função*, chamado na literatura de *assinatura*, onde cada $f \in \Sigma$ está associado com um inteiro não negativo n, chamado aridade de f e denotado por $arity(f) = n$, ou seja, $arity : \Sigma \to \mathbb{N}$. Esta *função aridade* foi especificada na *sub-teoria* `arity` (Veja Figura 5.1.1) e é dada por

```
arity:  [symbol -> nat].
```

Denotamos o conjunto de todos os símbolos de função de Σ cuja a aridade é $n \geq 0$ por $\Sigma^{(n)}$, e chamamos os elementos de $\Sigma^{(0)}$ de *símbolos de constante*.

Definição 5.2.1: Sejam Σ uma assinatura e V um conjunto arbitrário de símbolos de variável, ou simplesmente *variáveis*. Definimos indutivamente o conjunto de *termos* $T(\Sigma, V)$ como segue:

(a) Toda variável é um termo, ou seja, $V \subseteq T(\Sigma, V)$;

(b) Se f é um símbolo de função de aridade $n \geqslant 0$ ($f \in \Sigma^{(n)}$) e $t_1, \cdots, t_n \in T(\Sigma, V)$, então $f(t_1, \cdots, t_n)$ é um termo, ou seja, $f(t_1, \cdots, t_n) \in T(\Sigma, V)$. A seqüência de termos t_i é chamada os *argumentos* do termo $f(t_1, \cdots, t_n)$, e o símbolo de função f é a *cabeça* ou *raiz*.

O PVS não permite definições de tipos recursivos, e para contornar esta situação o PVS oferece o mecanismo primitivo `DATATYPE`, como vimos na Seção 3.5. Assim, especificamos o conjunto de termos $T(\Sigma, V)$ como um `DATATYPE`, chamado `term`, como mostrado na Tabela 5.2.1.

Tabela 5.2.1: Especificação recursiva de termos como Abstract DataTypes

```
term[variable: TYPE+, symbol: TYPE+] : DATATYPE
 BEGIN

  IMPORTING arity[symbol]

  vars(v: variable): vars?
  app(f:symbol, args:{args:finite_sequence[term] | args`length=arity(f)}): app?

 END term
```

Observe que a teoria `term` possui dois construtores, a saber: o construtor `vars` o qual recebe um único argumento de tipo `variable`, ou seja, variável; e o construtor `app` que recebe dois argumentos onde o primeiro deles é de tipo `symbol`, ou seja, um símbolo de função, e o segundo é de tipo `finite_sequence` instânciado com o tipo `term`, ou seja, uma seqüência finita de termos. Observe ainda que o argumento `args`, além de promover a chamada recursiva do `DATATYPE`, garante que o símbolo de função `f` esta sendo aplicado a um número correto de argumentos, ou seja, a aridade do símbolo de função `f` é exatamente igual ao comprimento da seqüência finita `args`.

Para distinguir, durante o desenvolvimento da *teoria* `trs` de forma mais clara, os termos que são do tipo aplicação dos termos que são do tipo variável, especificamos na *sub-teoria* `variables_term` (Veja Figura 5.1.1) o conjunto `V` de todos os termos que são de tipo variável, ou seja,

```
V: set[term] = {x:term | vars?(x)}.
```

Além disso, assumimos que o conjunto `V` é um conjunto infinito e enumerável, ou seja,

```
var_countable:  ASSUMPTION is_countably_infinite(V).
```

onde `is_countably_infinite` é um predicado pré-construído no PVS e que pode ser encontrado na sua biblioteca.

5.3 Posições

A *sub-teoria* `positions` (Veja Figura 5.1.1) contém a especificação do *conjunto de posições* de um termo, de acordo com a Definição 5.3.1, bem como a especificação de posições paralelas e a formalização de algumas propriedades envolvendo posições.

Posições de um termo são seqüências finitas de números naturais não-nulos, denotadas por $n_1 n_2 \cdots n_k$. Além disso, usamos a notação ip para representar a concatenação do número natural não-nulo i com a seqüência finita p, ou seja, se $p = n_1 n_2 \cdots n_k$ então $ip = i n_1 n_2 \cdots n_k$. De forma mais geral, definimos a concatenação de duas seqüências finitas $p = n_1 \cdots n_k$ e $q = m_1 \cdots m_k$ como sendo $pq = n_1 \cdots n_k m_1 \cdots m_k$, e especificada tal como apresentado na Tabela 5.1.1.

Definição 5.3.1: Seja Σ uma assinatura, V um conjunto de variáveis, com $\Sigma \cap V = \emptyset$, e $t \in T(\Sigma, V)$. Então definimos o conjunto de posições do termo t, denotado por $Pos(t)$, indutivamente como segue:

(a) Se $t = x \in V$, então $Pos(t) := \epsilon$, onde ϵ denota a posição vazia (seqüência vazia), chamada de *posição raiz*.

(b) Se $t = f(t_1, \cdots, t_n)$, então

$$Pos(t) := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in Pos(t_i)\}$$

Definimos ainda uma ordem parcial $\leq$ sobre o conjunto das posições de um termo dada por:

$$p \leq q \textit{ se, e somente se, existe uma posição } p' \textit{ tal que } pp' = q$$

Além disso, dizemos que duas posições p e q são *paralelas*, e denotamos por $p \parallel q$, se, e somente se, $p \not\leq q$ e $q \not\leq p$.

A especificação do conjunto de posições em PVS, chamado `positionsOF`, é feita recursivamente como a maioria dos conceitos da teoria TRS e como sugere a Definição 5.3.1. Nesta especificação usamos como medida para terminação a ordem bem-fundada `<<` que é

gerada automaticamente pela *operação typechecking* da teoria `term` (Ver Seção 3.1 e [50]), tal como apresentado na Tabela 5.3.1.

Tabela 5.3.1: Conjunto de Posições de um Termo

```
positionsOF(t: term): RECURSIVE positions =
  (CASES t OF
     vars(t): only_empty_seq,
     app(f, st): IF length(st) = 0
                 THEN only_empty_seq
                 ELSE union(only_empty_seq,
                            IUnion((LAMBDA (i: upto?(length(st))):
                                    catenate(i, positionsOF(st(i-1)) ))))
                 ENDIF
   ENDCASES)
MEASURE t BY <<
```

Na especificação apresentada na Tabela 5.3.1 usamos os construtores `only_empty_seq` que constitui um conjunto unitário contendo a seqüência vazia, e `catenate` que insere um número natural não-nulo na primeira posição de cada seqüência de um dado conjunto de seqüências finitas tal como mostra a Tabela 5.3.2.

Tabela 5.3.2: Propriedades sobre posições de um termo

```
position: TYPE = finseq[posnat]
positions: TYPE = set[position]

only_empty_seq: positions = {x: position | x = empty_seq}

catenate(i: posnat, s: positions): positions  =
        {seq: position | EXISTS (x: position): (member(x,s) AND
                                                seq = add_first(i,x))}

<=(p, q): bool = (EXISTS (p1: position): q = p o p1)

parallel(p, q): bool = (NOT p <= q) & (NOT q <= p)

positions_of_terms_finite : LEMMA is_finite(positionsOF(t))

closed_positions: LEMMA (positionsOF(t)(q) & p <= q) => positionsOF(t)(p)

not_var: LEMMA (positionsOF(t)(p) & p = add_first(i, q)) => app?(t)
```

Na *sub-teoria* `positions` também especificamos uma ordem parcial `<=` sobre o conjunto de posições, o conceito de posições paralelas onde a concatenação de duas seqüências p, q é representada por `p o q`, e formalizamos algumas propriedades básicas tais como: o conjunto de posições de um termo é fechado com relação a `<=` (`closed_positions`); e que o conjunto de posições de um termo é finito (`positions_of_terms_finite`). Veja Tabela 5.3.2 e observe que `position` é um tipo composto de seqüências finitas de números naturais não-nulos e que `positions` é um tipo composto por conjuntos de `position`.

5.4 Subtermos

Na *sub-teoria* `subterm` (Veja Figura 5.1.1) especificamos indutivamente as noções de *subtermos* de um dado termo de forma análoga à definição a seguir.

Definição 5.4.1: Sejam s, t termos e $p \in Pos(s)$.

(i) O *subtermo* de s na posição p, denotado por $s|_p$, é definido indutivamente sobre o comprimento de p:

$$\begin{array}{rcl} s|_\epsilon & := & s \\ f(s_1, ..., s_n)|_{iq} & := & s_i|_q \ . \end{array}$$

É fácil ver que se $p \in Pos(s)$ e $p = iq$ então s é da forma $s = f(s_1, ..., s_n)$ com $i \leq n$. Ressaltamos que esta afirmação foi provada mecânicamente na *sub-teoria* `positions` tal como mostra a Tabela 5.3.2.

(ii) O *conjunto das variáveis que ocorrem em* s, é denotado por $Var(s)$ e definido como segue:

$$Var(s) = \{x \in V \ \mid \ \exists p \in Pos(s) : s|_p = x\}$$

Na Tabela 5.4.1, apresentamos a especificação recursiva da noção de subtermo, chamada `subtermOF`. O construtor `subtermOF` recebe como entrada um termo `t` e uma posição `p` de `t`, e retorna o subtermo de `t` na posição `p`. Observe que dessa forma um subtermo de um dado termo é unicamente determinado pela sua posição.

Na Tabela 5.4.2 e daqui por diante, `positions?(t)` representa o tipo composto por todas as posições do termo `t`.

Tabela 5.4.1: Subtermos de um dado termo

```
subtermOF(t: term, (p: positions?(t))): RECURSIVE term =
      (IF length(p) = 0
       THEN t
       ELSE LET st = args(t),
                 i = first(p),
                 q = rest(p)  IN
            subtermOF(st(i-1), q)
       ENDIF)
MEASURE length(p)
```

Na *sub-teoria* `subterm` além de especificar recursivamente a noção de subtermo, cuja medida é o comprimento da posição `p`, especificamos o conjunto de variáveis de um termo `t` e o *conjunto das posições* de um dado termo `t` onde ocorre uma dada variável `x`, respectivamente denotados por `Vars(t)` e `Pos_var(t, x)` e apresentados na Tabela 5.4.2. Também na Tabela 5.4.2 apresentamos a formalização do Lema 5.4.2.

Lema 5.4.2: Se $pq \in Pos(s)$, então:

1. $q \in Pos(s|_p)$.
2. $s|_{pq} = (s|_p)|_q$.

Demonstração: Ambas as demonstrações são por indução no comprimento de p. Como exemplo, apresentamos a prova, por indução no comprimento de p, do item *2*:

BI: Seja $p = \epsilon$. Assim $pq = q$ e $s|_p = s$. Portanto, $s|_{pq} = s|_q = (s|_p)|_q$.

PI: Suponhamos que a afirmação é verdadeira para toda posição p' de s cujo comprimento é menor do que o comprimento de p.

Vamos assumir agora que $p = ip'$. Uma vez que $ip'q \in Pos(s)$ temos que s é da forma $s = f(s_1, ..., s_n)$ com $i \leq n$. Logo por definição,

$$s|_{pq} = s|_{ip'q} = s_i|_{p'q} \ .$$

Da Hipotése de Indução temos que

$$s_i|_{p'q} = (s_i|_{p'})|_q \ .$$

Finalmente, por definição, $s_i|_{p'} = s|_{ip'} = s|_p$, o que completa a demonstração. ■

O Lema 5.4.2 foi formalizado em forma de dois lemas, `pos_subterm_ax` e `pos_subterm`, e as técnicas usadas para provar mecânicamente estes resultados são as mesmas usadas na prova (analítica) dada acima, ou seja, indução no comprimento da posição `p`.

Tabela 5.4.2: Propriedades básicas de subtermos

```
Vars(t): set[(V)] = {x: (V) |  EXISTS (p: positions?(t)): subtermOF(t,p) = x}

Pos_var(t, x): positions = {p: positions?(t) | subtermOF(t,p) = x}

pos_subterm_ax: LEMMA positionsOF(t)(p o q) => positionsOF(subtermOF(t,p))(q)

pos_subterm: LEMMA  positionsOF(t)(p o q) =>
                              subtermOF(t, p o q) = subtermOF(subtermOF(t,p),q)
```

Para realizar e controlar provas indutivas sobre o comprimento de uma seqüência (posição), o PVS oferece a estratégia, dentre outras, conhecida como `measure-induct+`. Esta estratégia toma como argumentos um *medida* e uma *variável de indução* para a qual a *medida* está definida. Assim, para realizar as provas mecânicas dos lemas apresentados na Tabela 5.4.2 invocamos a estratégia `measure-induct+` instânciada adequadamente para realizar tais provas por indução sobre o comprimento de uma seqüência `p`, ou seja, `(measure-induct+ "length(p)" "p")` onde a medida é o comprimento da seqüência `length(p)` e a variável de indução é a seqüência `p`.

Por exemplo, para provar o lema `pos_subterm_ax` invocamos a estratégia `(measure-induct+ "length(p)" "p")` e obtemos como resultado um sequente da seguinte forma:

```
{-1}  FORALL (y: position):
        FORALL (q: position, t: term):
          length(y) < length(p) IMPLIES
           positionsOF(t)(y o q) => positionsOF(subtermOF(t, y))(q)
  |-------
{1}   FORALL (q: position, t: term):
        positionsOF(t)(p o q) => positionsOF(subtermOF(t, p))(q)
```

Note que a estratégia gera no antecedente uma hipótese de indução (`{-1}`) sobre o comprimento da seqüência `p`. Isto é, se para toda seqüência `y` de comprimento menor que o comprimento da seqüência `p` vale a propriedade `positionsOF(subtermOF(t, y))(q)`, então devemos provar que para todo `p` vale a propriedade `positionsOF(subtermOF(t, p))(q)`. Observe também que o passo indutivo (`length(p) = 0`) segue trivialmente pela própria definição de subtermo.

5.5 Troca de Subtermos

Na *sub-teoria* `replacement` (Veja Figura 5.1.1) especificamos indutivamente a noção de *trocar* um subtermo de um dado termo s por um outro termo t, tendo como base a definição a seguir.

Definição 5.5.1: Sejam s, t termos e $p \in Pos(s)$. O resultado de *trocar* o subtermo de s na posição p por t é denotado por $s[p \leftarrow t]$ e obtido indutivamente como segue:

$$\begin{array}{rcl} s[\epsilon \leftarrow t] & := & t \\ f(s_1, ..., s_n)[iq \leftarrow t] & := & f(s_1, ..., s_i[q \leftarrow t], ..., s_n). \end{array}$$

Como mostra a Tabela 5.5.1, o construtor `replaceTerm`, cuja medida para terminação é o comprimento da posição `p`, contempla a Definição 5.5.1. Este construtor possui três argumentos como entrada: O terceiro argumento determina o subtermo (posição) do segundo argumento que vai ser trocado pelo termo fornecido no primeiro argumento.

Tabela 5.5.1: Especificação recursiva do construtor `replaceTerm`.

```
replaceTerm(t: term, s: term, (p: positions?(s))): RECURSIVE term =
   (IF length(p) = 0
    THEN t
    ELSE LET st = args(s),
             i = first(p),
             q = rest(p),
           rst = replace(replaceTerm(t, st(i-1), q), st,i-1) IN
         app(f(s), rst)
    ENDIF)
 MEASURE length(p)
```

o construtor `replaceTerm` desempenha vários papéis fundamentais no decorrer do desenvolvimento da *teoria* `trs`, por exemplo, este construtor será fundamental para a caracterização da noção de *relação de redução* e da noção de *pares críticos* como veremos, respectivamente, nas Seções 5.9 e 5.10.

Uma vez especificado o construtor `replaceTerm`, formalizamos propriedades básicas e utéis que permitem manipular o seu uso. Tais propriedades são apresentadas no lema a seguir, e as formalizações de algumas delas são apresentadas na Tabela 5.5.2. Estas propriedades são fundamentais no desenvolvimento de muitos resultados da *teoria* `trs`, como por exemplo na formalização do Teorema dos Pares Críticos de Knuth-Bendix, como veremos na Seção 5.10.

Lema 5.5.2: Sejam s, t, r termos e p, q posições.

1. Se $p \in Pos(s)$, então
   (a) $p \in Pos(s[p \leftarrow t])$
   (b) $s[p \leftarrow t]|_p = t$
   (c) $s[p \leftarrow s|_p] = s$
   (d) Se $q \in Pos(t)$, então $pq \in Pos(s[p \leftarrow t])$
2. Seja $p \parallel q$, então temos
   (a) Se $p \in Pos(s)$ e $q \in Pos(s)$, então $q \in Pos(s[p \leftarrow t])$
   (b) Se $p \in Pos(s)$ e $q \in Pos(s[p \leftarrow t])$, então $q \in Pos(s)$
3. Se $p \in Pos(s)$ e $q \in Pos(t)$, então
   (a) $s[p \leftarrow t]|_{pq} = t|_q$ submersão
   (b) $s[p \leftarrow t][pq \leftarrow r] = s[p \leftarrow t[q \leftarrow r]]$ associatividade
4. Se $pq \in Pos(s)$, então
   (a) $s[pq \leftarrow t]|_p = (s|_p)[q \leftarrow t]$ distributividade
   (b) $s[pq \leftarrow t][p \leftarrow r] = s[p \leftarrow r]$ dominância

5. Se $p, q \in Pos(s)$ e $p \parallel q$, então

   (a) $s[p \leftarrow t]|_q = s|_q$ persistência

   (b) $s[p \leftarrow t][q \leftarrow r] = s[q \leftarrow r][p \leftarrow t]$ comutatividade

Demonstração: Os itens 1, 3.(b) e 4 são demonstrados por indução no comprimento de p. Já os itens 2.(a) e 5 são demonstrados por indução no comprimento de q. O item 3.(a) segue diretamente da aplicação do item 2 do lema 5.4.2, e dos itens 1.(b) e 1.(d) acima. Por último, a prova do item 2.(b) segue por indução na estrutura do termo s, a qual apresentamos a seguir:

BI: Suponhamos que s seja uma variável. Assim, $p = \epsilon$ e $s[p \leftarrow t] = t$. Temos então dois casos: se t é uma variável ou uma constante então $q = \epsilon$ e, portanto, $q = p = \epsilon \in Pos(s)$. Porém, se $t = g(t_1, \cdots, t_m)$ temos: se $q = \epsilon$ nada a fazer. Se $q = iq'$ então $q = pq$ o que contraria a hipótese $p \parallel q$.

PI: Suponhamos que $s = f(s_1, \cdots, s_n)$ para $n \geq 0$. Se $n = 0$ então s é uma constante, $p = \epsilon$, e a prova segue análoga à **BI**. Caso contrário, $n > 0$, então devemos considerar duas situações: se $p = \epsilon$ e/ou $q = \epsilon$ então nada a fazer. Se $p, q \neq \epsilon$ consideramos dois casos: se $p = ip'$ e $q = iq'$, pela hipótese de indução, temos que $q' \in Pos(s_i)$ e consequentemente $q \in Pos(s)$. Finalmente, se $p = ip'$ e $q = jq'$, com $i \neq j$, o resultado segue por definição. ■

Para formalizar as propriedades apresentadas no Lema 5.5.2 usamos não mais do que as técnicas apresentadas na prova (analítica) dada acima. Por exemplo, para formalizar a propriedade `lemma1B` apresentada na Tabela 5.5.2 e correspondente à propriedade 2.(b) do Lema 5.5.2 usamos indução na estrutura do termo `s`.

O *esquema de indução estrutural* que usamos é automaticamente gerado pela *operação typechecking* (Ver Seção 3.1) quando aplicada a um `DATATYPE` [50]. Por exemplo, para a *sub-teoria* `term` foi gerado o esquema de indução estrutural sobre os termos tal como apresentado na Tabela 5.5.3.

Tabela 5.5.2: Propriedades básicas do construtor `replaceTerm`.

```
lemmaR6: LEMMA positionsOF(s)(p o q) =>
               replaceTerm(r,replaceTerm(t,s,p o q),p) = replaceTerm(r,s,p)

lemmaR7: LEMMA positionsOF(s)(p) & positionsOF(s)(q) & parallel(p,q) =>
               subtermOF(replaceTerm(t, s, p), q) = subtermOF(s,q)

lemmaR8: LEMMA positionsOF(s)(p) & positionsOF(s)(q) & parallel(p,q) =>
     replaceTerm(r,replaceTerm(t,s,p),q) = replaceTerm(t,replaceTerm(r,s,q),p)

lemma1B: LEMMA positionsOF(s)(p) & positionsOF(replaceTerm(t, s, p))(q) &
               parallel(p, q) => positionsOF(s)(q)
```

O esquema de indução apresentado na Tabela 5.5.3 nos diz que se todas as variáveis `v` satisfazem a propriedade `p`, e se para todos os termos do tipo aplicação, `app(f, args)`, os seus sub-termos próprios satisfazem a propriedade `p` e implica que `app(f, args)` também satisfazem a propriedade `p`, então todo termo `t` satisfaz a propriedade `p`.

Tabela 5.5.3: Esquema de indução estrutural para termos.

```
term_induction: AXIOM
  FORALL (p: [term -> boolean]):
    ((FORALL (v: variable): p(vars(v))) AND
      (FORALL (   f: symbol,
                args: {args: finite_sequence[term] | args`length = arity(f)}):
                   (FORALL (x: below[args`length]): p(args`seq(x)))
                       IMPLIES
                            p(app(f, args))))
     IMPLIES (FORALL (t: term): p(t))
```

5.6 Compatibilidade

A especificação da Definição 5.6.1 e a formalização do Lema 5.6.2, apresentados a seguir, estão contidas na *sub-teoria* `compatibility` (Veja Figura 5.1.1).

Definição 5.6.1: Seja $\equiv$ uma relação binária sobre $T(\Sigma, V)$.

(a) A relação $\equiv$ é *fechada para operações* se, e somente se, $s_1 \equiv t_1, \cdots, s_n \equiv t_n$ implica

$f(s_1, \cdots, s_n) \equiv f(t_1, \cdots, t_n)$ para todo $n \geq 0$, $f \in \Sigma^{(n)}$ e $s_1, \cdots, s_n, t_1, \cdots, t_n \in T(\Sigma, V)$.

(b) A relação $\equiv$ é *compatível com operações* se, somente se, $s \equiv t$ implica

$$f(s_1, \cdots, s_{i-1}, s, s_{1+1}, \cdots, s_n) \equiv f(s_1, \cdots, s_{i-1}, t, s_{i+1}, \cdots, s_n)$$

para todo $n \geq 0$, $f \in \Sigma^{(n)}$, $i = 1, \cdots, n$ e $s_1, \cdots, s_{i-1}, t, s_{i+1}, \cdots, s_n \in T(\Sigma, V)$.

(c) A relação $\equiv$ é *compatível com contexto* se, e somente se, $s \equiv s'$ implica $t[p \leftarrow s] \equiv t[p \leftarrow s']$ para todo $t \in T(\Sigma, V)$ e $p \in Pos(t)$.

O seguinte lema é uma conseqüência imediata da definição acima.

Lema 5.6.2: Seja $\equiv$ uma relação binária sobre $T(\Sigma, V)$. A relação $\equiv$ é compatível com operações se, e somente se, ela é compatível com contexto.

Demonstração: A direção $\Leftarrow$ é óbvia, e a direção $\Rightarrow$ é mostrada por indução no comprimento de p. ∎

Na Tabela 5.6.1 apresentamos somente as especificações de compatibilidade com contexto e operações (`comp_cont?` e comp_op?) de uma relação binária `R`, e a especificação do lema acima.

Tabela 5.6.1: Relações compatíveis com contexto e operações

```
comp_op?(R): bool =
    FORALL (n: nat, i: below[n], (f: arity(n)), (st1: fset.fs_len(n)), t):
       LET s = st1(i), st2 = replace(t, st1, i) IN
         R(s,t) =>  R(app(f,st1), app(f,st2))

comp_cont?(R): bool = FORALL (p: position), t:
      positionsOF(t)(p) =>
        (FORALL r, s: R(r,s) => R(replaceTerm(r, t, p), replaceTerm(s, t, p)))

lemmaRED2: LEMMA comp_op?(R) <=> comp_cont?(R)
```

5.7 Substituições, Renomeamentos e Unificadores

Na *sub-teoria* `substitution` (Veja Figura 5.1.1) especificamos o que venha a ser uma *substituição*, um *renomeamento*, e outros conceitos tais como: *domínio e imagem* de uma substituição, *unificador*, e *unificador mais geral*. Como veremos mais adiante, também, nesta *sub-teoria* formalizamos algumas propriedades referentes à substituições necessárias para o desenvolvimeto da *teoria* `trs`.

5.7.1 Substituições e Renomeamentos

Definição 5.7.1: Seja Σ uma assinatura e V um conjunto infinito e enumerável de variáveis.

(a) Uma *substituição* é uma função $\sigma : V \to T(\Sigma, V)$ tal que $\sigma(x) \neq x$ somente para um número finito de xs.

(b) O conjunto das variáveis x tais que $\sigma(x) \neq x$ é finito, chamado de *Domínio de* σ, e denotado por $Dom(\sigma)$. Em outras palavras,

$$Dom(\sigma) := \{x \in V \mid \sigma(x) \neq x\}.$$

(c) A *imagem* de uma substituição σ, denotada por $Ran(\sigma)$, é definido como:

$$Ran(\sigma) := \{\sigma(x) \mid x \in Dom(\sigma)\}$$

(d) Um *renomeamento* é uma substituição injetiva ρ tal que $Ran(\rho) \subseteq V$, ou seja, ρ é uma bijeção sobre V. Em particular, ρ é uma bijeção entre $Dom(\rho)$ e $Ran(\rho)$.

Em PVS, inicialmente especificamos funções `sig:(V) -> term`, onde `(V)` é o tipo gerado pelo conjunto dos termos variáveis `V = {x:term | vars?(x)}`. Depois especificamos o domínio e a imagem de uma função `sig`. E então, por meio de uma booleana, especificamos o que é uma substituição, como podemos ver na Tabela 5.7.1, a qual é um pequeno fragmento do que foi especificado e formalizado na *sub-teoria* `substituion`.

Também na Tabela 5.7.1 apresentamos a especificação da noção de renomeamento de variáveis, denotado por `Ren?`, e cujo argumento `sigma` é uma substituição.

Ressaltamos que, em geral, usar índices de de Bruijn em vez de variáveis com nomes evita dificuldades em construir renomeamentos de variáveis bijetivos. Por exemplo, em [46] McKinna e Pollack representam os renomeamentos como uma lista de pares de variáveis, e usam um operador para obter novos nomes de variáveis. Além disso, McKinna e Pollack usam uma representação que eles chamam de *tricky* para representar a ação dos renomeamentos. No entanto, segundo eles, esse *tricky* dificulta a construção de renomeamentos bijetivos. Assim como Ford e Mason [18], preferimos uma representação próxima da representação natural matemática sempre que possível. E por isso, incorporando diretamente na especificação de renomeamento a condição de ser bijetivo.

Tabela 5.7.1: Substituição, Renomeamento, Domínio e Imagem

```
sig: VAR [(V) -> term]

Dom(sig): set[(V)] = {x: (V) | sig(x) /= x}

Ran(sig): set[term] =
                 {y: term | EXISTS (x: (V)): member(x, Dom(sig)) &  y = sig(x)}

Sub?(sig): bool = is_finite(Dom(sig))

Ren?(sigma): bool =  subset?(Ran(sigma),V) &
                        (bijective?[(Dom(sigma)),(Ran(sigma))])(sigma)
```

Na seqüência, estendemos homeomorficamente uma substituição σ para o conjunto $T(\Sigma, V)$.

Definição 5.7.2: Seja σ uma substituição. Então definimos $\widehat{\sigma} : T(\Sigma, V) \to T(\Sigma, V)$ assim

$$\widehat{\sigma}(s) := \begin{cases} \sigma(x), & \text{se} \quad s = x \in V \\ f(\widehat{\sigma}(s_1), \cdots, \widehat{\sigma}(s_n)), & \text{se} \quad s = f(s_1, \cdots, s_n) \end{cases}$$

Na Tabela 5.7.2 apresentamos a especificação recursiva de $\widehat{\sigma}(s)$, que em PVS é denota por `ext(sigma)` e que recebe como entrada um termo `t`.

Tabela 5.7.2: Extendendo uma substituição para $T(\Sigma, V)$

```
ext(sigma)(t): RECURSIVE term =
   CASES t OF
     vars(t): sigma(t),
  app(f, st): IF length(st) = 0
              THEN t
              ELSE LET sst = (# length := st`length,
                               seq := (LAMBDA (n: below[st`length]):
                                                ext(sigma)(st(n)))#) IN
                   app(f, sst)
              ENDIF
   ENDCASES
MEASURE t BY <<
```

Uma vez especificada a noção de estender uma substituição para $T(\Sigma, V)$, especificamos a operação *composição* de duas substituições.

Definição 5.7.3: A *composição* $\sigma\tau$ de duas substituições σ e τ é definida como

$$\sigma\tau(x) := \widehat{\sigma}(\tau(x)).$$

É fácil ver que $\sigma\tau : V \to T(\Sigma, V)$ e que $\sigma\tau(x) = x$ para todo $x \in V - (Dom(\sigma) \cup Dom(\tau))$, ou seja, que $\sigma\tau$ é também uma substituição. Além disso é possível provar que a operação composição de substituições é associativa, e que a extensão da composição $\sigma\tau$ é exatamente igual à composições das extensões, ou melhor, $\widehat{\sigma\tau} = \widehat{\sigma}\widehat{\tau}$.

A Tabela 5.7.3 apresenta a especificação da composição de duas substituições, assim como, as formalizações das propriedades citadas acima.

Tabela 5.7.3: Composição de substituições e algumas propriedades

```
comp(sigma, tau)(x: (V)): term = ext(sigma)(tau(x))

subs_o: LEMMA Sub?(comp(sigma, tau))

o_ass: LEMMA comp(comp(sigma, delta), tau) = comp(sigma, comp(delta, tau))

ext_o: LEMMA ext(comp(sigma, tau)) = ext(sigma) o ext(tau)
```

As provas dos lemas `subs_o` e `o_ass` são simples e, de forma geral, são obtidas expandindo as definições envolvidas. Já a prova do lema `ext_o` se mostra um pouco mais interessante. Observe que na verdade provar o lema `ext_o` significa provar a igualdade entre duas funções, ou seja, devemos provar que para todo termo `t` temos que

```
ext(comp(sigma, tau))(t) = ext(sigma) o ext(tau)(t).
```

Para obter esta última igualdade a partir do lema `ext_o`, usamos o comando (regra) de prova `(decompose-equality)` o qual decompõe uma igualdade antecedente ou consequente de funções, `f = g`, retornando e inserindo no sequente corrente, a quantificação universal `(FORALL x: f(x) = g(x))`. Em outras palavras, o comando `(decompose-equality)` aplica um esquema de extensionalidade para provar a igualdade, dentre outras, entre duas funções. Agora, para provar a igualdade acima usamos indução em `t`, ou seja, usamos um esquema de *indução estrutural* sobre `t`.

Definição 5.7.4: Uma substituição σ é *mais geral* do que uma substituição τ se existe uma substituição δ tal que $\tau = \delta\sigma$.

Na Tabela 5.7.4 apresentamos a especificação da relação $\lesssim$, denotada por `<=`, e a formalização de que a relação $\lesssim$ é uma pré-ordem.

Tabela 5.7.4: Definição de substituição mais geral

```
<=(sigma, tau): bool = EXISTS delta: tau = comp(delta, sigma)

mg_po: LEMMA preorder?(<=)
```

5.7.2 Unificadores

A noção de unificador é especificada na *sub-teoria* `substitution`, e a existência e unicidade de *unificadores mais gerais* é tratada na *teoria* `trs` de forma axiomática.

Definição 5.7.5: Um *problema de unificação* é um conjunto de equações $S = \{s_1 =^? t_1, \cdots, s_n =^? t_n\}$. Um *unificador* de S é uma substituição σ tal que $\sigma(s_i) = \sigma(t_i)$ para

todo $i = 1, \cdots, n$. Denotamos o conjunto de todos os unificadores de S por $U(S)$. Além disso, dizemos que uma substituição σ é um *unificador mais geral* (mgu) de S se σ satisfaz as seguintes condições:

1. $\sigma \in U(S)$;
2. $\forall\, \tau \in U(S).\ \sigma \lesssim \tau$.

Na *sub-teoria* `substitution` consideramos o conjunto S como sendo unitário, ou seja, $S = \{s =^? t\}$ e a partir daí especificamos todos os conceitos apresentados na definição 5.7.5, como mostra a Tabela 5.7.5.

Tabela 5.7.5: Unificador e unificador mais geral

```
unifier(sigma)(s,t): bool = ext(sigma)(s) = ext(sigma)(t)

unifiable(s,t): bool = EXISTS sigma: unifier(sigma)(s,t)

U(s,t): set[Sub] = {sigma: Sub | unifier(sigma)(s,t)}

mgu(sigma)(s,t): bool = member(sigma, U(s,t)) &
                        FORALL tau: member(tau, U(s,t)) => sigma <= tau
```

5.8 Regras de Reescrita

Nesta seção introduzimos a noção do principal objeto de estudo deste capítulo: *Sistemas de Reescrita de Termos* [4].

Definição 5.8.1: Um *Sistema de Reescrita de Termos* é um conjunto E de pares de termos $(l, r) \in T(\Sigma, V) \times T(\Sigma, V)$ tais que l não é uma variável e $Var(r) \subseteq Var(l)$. Um par de termos (l, r) satisfazendo as condições anteriores é chamado de *regra de reescrita* e denotado por $l \to r$ em vez de (l, r).

Na *sub-teoria* `rewrite_rules` (Veja Figura 5.1.1) especificamos o que venha a ser uma regra de reescrita tal como mostrado na Tabela 5.8.1.

Tabela 5.8.1: Regras de Reescrita

```
rewrite_rule?(l,r): bool = (NOT vars?(l)) & subset?(Vars(r), Vars(l))

rewrite_rule: TYPE = (rewrite_rule?)
```

Na Tabela 5.8.1 apresentamos a construção do tipo `rewrite_rule`, gerado pelas regras de reescrita `rewrite_rule?`. Este tipo é o fundamento para a construção de um Sistema de Reescrita de Termos `E`, ou seja,

```
E: VAR set[rewrite_rule]
```

e é a base para a caracterização de uma *relação de redução* como mostra a próxima seção.

5.9 Relação de Redução

Na *sub-teoria* `reduction` (Veja Figura 5.1.1) especificamos a caracterização de uma *relação de redução* induzida por um conjunto de regras de reescrita E. Após especificar a propriedade *fechada para substituição*, formalizamos que uma relação de redução é uma *relação de reescrita*.

Definição 5.9.1: Sejam E um sistema de reescrita de termos e $\equiv$ uma relação binária sobre $T(\Sigma, V)$.

(a) A relação $\equiv$ é *fechada para substituições* se, e somente se, $s \equiv t$ implica $\sigma(s) \equiv \sigma(t)$ para todo $s, t \in T(\Sigma, V)$ e toda substituição σ.

(b) Uma relação sobre $T(\Sigma, V)$ é uma *relação de reescrita* se, e somente se, ela é compatível com operações e fechada para substituições.

(c) Definimos a *Relação de Redução* $\rightarrow_E \subseteq T(\Sigma, V) \times T(\Sigma, V)$ como segue:

$$s \rightarrow_E t \;\; \Leftrightarrow \;\; \exists (l, r) \in E, p \in Pos(s), \sigma \in Sub. \;\; s|_p = \sigma(l) \;\text{ e }\; t = s[p \leftarrow \sigma(r)]$$

Como conseqüência da definição de $\rightarrow_E$ podemos provar o Lema 5.9.2, cuja demonstração pode ser encontrada em [4] e omitiremos aqui, e que nos atesta que a relação de redução $\rightarrow_E$ é na verdade uma relação de reescrita. Além disso, segue imediatamente dos Lemas 5.6.2 e 5.9.2 que a relação de redução $\rightarrow_E$ é compatível com o contexto.

Lema 5.9.2: Seja E um sistema de reescrita de termos. A relação de redução $\rightarrow_E$ é fechada para substituições e compatível com operações.

Tabela 5.9.1: Relação de Redução

```
reduction?(E)(s,t): bool =
     EXISTS ( (e | member(e, E)), sigma, (p: positions?(s))):
                       subtermOF(s, p) = ext(sigma)(lhs(e)) &
                                         t = replaceTerm(ext(sigma)(rhs(e)), s, p)

close_subs?(R): bool = FORALL s, t, sigma:
                           R(s,t) => R(ext(sigma)(s),ext(sigma)(t))

subs_op: LEMMA close_subs?(reduction?(E)) & comp_op?(reduction?(E))
```

Na Tabela 5.9.1 apresentamos a especificação de uma relação de redução e a formalização do Lema 5.9.2. Observe que, como mencionamos anteriormente, o construtor `replaceTerm` é fundamental para a caracterização de uma relação de redução. Além disso, na Tabela 5.9.1, o par `e:[term,term]` representa uma regra de reescrita onde `lhs(e):term = e‘1` e `rhs(e):term = e‘2`, ou seja, `lhs` e `rhs` representam, respectivamente, o lado esquerdo (primeira coordenada) e o lado direito (segunda coordenada) de `e`.

Note que já especificamos o principal objeto de manipulação da teoria de TRS que são os termos. Especificamos o que venha ser o conjunto de posições de um termo o qual permite fazer manipulações sobre os termos, bem como caracterizar um subtermo de um termo, o que também já foi especificado. Especificamos algumas ferramentas necessárias para manipular termos e subtermos, como por exemplo substituição e a noção de trocar um subtermo por algum outro termo. Também especificamos a noção de regras de reescrita e, é claro, a noção de relação de redução. Além disso, formalizamos várias propriedades referentes aos conceitos citados acima e apresentados em parte nas seções anteriores. Dessa forma já possuímos todos os ingredientes necessários para lidar com

propriedades e resultados gerais da teoria de TRS, como veremos na seção seguinte.

5.10 Formalização do Teorema dos Pares Críticos

Durante o desenvolvimento da *teoria* `trs` provamos vários lemas e propriedades sobre reescrita de termos, porém, o principal resultado que formalizamos, e que pode ser encontrado na *sub-teoria* `critical_pairs` (Veja Figura 5.1.1), foi o *Teorema dos Pares Critícos de Knuth-Bendix* o qual apresentamos na seqüência, depois de definir o que venha a ser um *par crítico*.

Definição 5.10.1: Sejam $l_i \to r_i$, $i = 1, 2$, duas regras de reescrita cujas variáveis foram renomeadas de forma que $Var(l_1) \cap Var(l_2) = \emptyset$. Seja $p \in Pos(l_1)$ tal que $l_1|_p$ não seja uma variável e $\sigma = mgu(l_1|_p, l_2)$. Neste caso, dizemos que a sobreposição de $l_2 \to r_2$ sobre $l_1 \to r_1$ na posição p determina um *par crítico* $\langle t_1, t_2 \rangle$, definido por

$$\begin{array}{lcl} t_1 & = & \sigma(r_1) \\ t_2 & = & \sigma(l_1)[p \leftarrow \sigma(r_2)]. \end{array}$$

Em geral, nos livros texto, assim como na definição acima, assume-se que as regras de reescrita são renomeadas de modo a obter que $Var(l_1) \cap Var(l_2) = \emptyset$. No entanto, na prática, para especificar a noção de par crítico foi necessário introduzir explicitamente um renomeamento para garantir tal condição, ou seja, na prática a definição analítica de par crítico não é tão explícita como a sua especificação. Assim, vemos que a noção de renomeamento é de extrema importância na especificação da noção de par crítico, e se tivessemos optado por usar índices de de Bruijn ao invés de variáveis com nomes, em nossa especificação, em algum momento teríamos que especificar a noção de renomeamento de uma forma concreta, e , é claro, a abordagem seria diferente.

Observe que na realidade, para garantir que $Var(l_1) \cap Var(l_2) = \emptyset$, é suficiente renomear apenas umas das regras, e foi assim que fizemos. Em outras palavras, introduzimos explicitamente na especificação de um par crítico um renomeamento ρ o qual renomeia as variáveis da regra de reescrita $l_2 \to r_2$ de forma a obter a condição $Var(l_1) \cap Var(l_2) = \emptyset$, tal como mostrado na Tabela 5.10.1.

Na Tabela 5.10.1, apresentamos a especificação da booleana `CP?` a qual verifica se dois termos `t1` e `t2` formam um par crítico. Na especificação, apresentada na Tabela 5.10.1, `E` é um conjunto de regras de reescrita, `sigma` é uma substituição, `rho` é um renomeamento, `e1` e `e2p` são regras de reescrita pertencentes a `E`, e `e2` é a regra de reescrita resultante da aplicação do renomeamento `rho` à regra `e2p`.

Tabela 5.10.1: Par Crítico

```
CP?(E)(t1, t2): bool =
   EXISTS (sigma,rho,
           (e1 | member(e1, E)),
           (e2p | member(e2p, E)),
           (p: positions?(lhs(e1)))):
         LET e2 = (# lhs := ext(rho)(lhs(e2p)), rhs := ext(rho)(rhs(e2p)) #) IN
              disjoint?(Vars(lhs(e1)),Vars(lhs(e2)))                          &
              NOT vars?(subtermOF(lhs(e1), p))                                &
              mgu(sigma)(subtermOF(lhs(e1), p), lhs(e2))                      &
              t1 = ext(sigma)(rhs(e1))                                        &
              t2 = replaceTerm(ext(sigma)(rhs(e2)), ext(sigma)(lhs(e1)), p)
```

A prova mecânica do *Teorema dos Pares Críticos de Knuth-Bendix*, apresentada a seguir, segue a estrutura da prova dada por Huet em [34]. Note que para formalizar este teorema foi necessário utilizar conceitos previamente especificados na *teoria* `ars`.

Teorema 5.10.2 [Teorema dos Pares Críticos de Knuth-Bendix]: Um sistema de reescrita de termos E é localmente confluente se, e somente se, todos os seus pares críticos são juntáveis.

Na Tabela 5.10.2 apresentamos a especificação do Teorema dos Pares Críticos de Knuth-Bendix em PVS e, fugindo um pouco da metodologia adotada até agora, em seguida apresentamos um esboço da sua formalização em PVS.

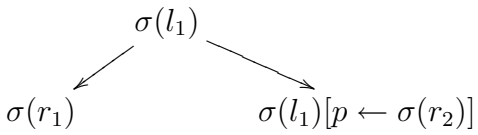
No que segue, estamos considerando a $\rightarrow$ como sendo a relação de redução induzida pelo conjunto de regras de reescrita `E`, isto é, `reduction?(E)`.

Iniciamos a formalização considerando a direção `=>`. Como todo par crítico é obtido

Tabela 5.10.2: Teorema dos Pares Críticos de Knuth-Bendix

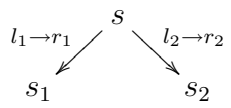
```
CP_Theorem: THEOREM
 FORALL E:
   LET RRE = reduction?(E) IN
   local_confluent?(RRE)
               <=>
(FORALL t1, t2: CP?(E)(t1, t2) => joinable?(RRE)(t1,t2))
```

de uma divergência da forma



a hipótese de ser localmente confluente nos leva imediatamente à juntabilidade dos pares críticos, ou seja, a direção `=>` é imediata.

Para formalizar a direção `<=`, assumimos que todo par crítico $\langle t_1, t_2 \rangle$ de E é juntável. Assim, seja s um termo arbitrário tal que



ou seja, existem posições $p_i \in Pos(s)$, regras $l_i \rightarrow r_i \in E$, e substituições σ_i tais que $s|_{p_i} = \sigma_i(l_i)$ e $s_i = s[p_i \leftarrow \sigma_i(r_i)]$.

Logo, afim de provar mecanicamente que s_1 e s_2 são juntáveis consideramos dois casos:

Caso 1: Suponhamos que as posições p_1 e p_2 são paralelas. Neste caso, a prova mecânica deste caso segue os seguintes passos: por persistência (Ver Lema 5.5.2) obtemos que $s_1|_{p_2} = \sigma_2(l_2)$ e que $s_2|_{p_1} = \sigma_1(l_1)$. Além disso, por comutatividade(Ver Lema 5.5.2) temos que $s_3 = s_1[p_2 \leftarrow \sigma_2(r_2)] = s_2[p_1 \leftarrow \sigma_1(r_1)]$. Portanto, s_1 e s_2 são juntáveis e a confluência local segue.

Caso 2: Suponhamos que $p_1 = p_2 p$ ou $p_2 = p_1 p$, para algum p possivelmente vazio. Sem perca de generalidade assumimos que $p_2 = p_1 p$.

Iniciamos a formalização deste caso estabelecendo as seguintes propriedades: 1. pelo Lema 5.4.2 obtemos $\sigma_1(l_1|_p) = \sigma_2(l_2)$; e 2. pela propriedade distributiva (Ver Lema 5.5.2) obtemos $s_2|_{p_1} = \sigma_1(l_1)[p \leftarrow \sigma_2(r_2)]$.

Em seguida, mostramos que existe um termo s_3 tal que $\sigma_1(r_1) \rightarrow s_3$ e $s_2|_{p_1} \rightarrow s_3$. Então, concluimos pela compatibilidade da relação $\rightarrow$ que s_1 e s_2 são juntáveis. Para isto consideramos dois casos:

Caso 2a: $p \in Pos(l_1)$, $l_1|_p$ não é uma variável e $\sigma_1(l_1|_p) = \sigma_2(l_2)$. Para contemplar este caso formalizamos o lema auxiliar `CP_lemma_aux1` tal como mostrado na Tabela 5.10.3. Este lema nos diz que a divergência $\sigma_1(r_1)$ e $s_2|_{p_1}$ é na verdade uma instância

Tabela 5.10.3: Sobreposição dos Lados Esquerdos das Regras de Reescrita

```
CP_lemma_aux1: LEMMA
 FORALL E, (e1 | member(e1, E)), (e2 | member(e2, E)), (p: position):
 ( positionsOF(lhs(e1))(p)                                                    &
   NOT vars?(subtermOF(lhs(e1), p))                                           &
   ext(sg1)(subtermOF(lhs(e1), p)) = ext(sg2)(lhs(e2)) )
        =>
         EXISTS t1, t2, delta:
           CP?(E)(t1, t2)                                                     &
           ext(delta)(t1) = ext(sg1)(rhs(e1))                                 &
         ext(delta)(t2) = replaceTerm(ext(sg2)(rhs(e2)), ext(sg1)(lhs(e1)), p)
```

de um certo par crítico $\langle t_1, t_2 \rangle$, o qual por hipótese é juntável, ou seja, existe um termo t_3 tal que $t_1 \rightarrow^* t_3$ e $t_2 \rightarrow^* t_3$. Assim, existe uma substituição δ tal que $\delta(t_1) = \sigma_1(r_1)$ e $\delta(t_2) = s_2|_{p_1}$. Logo fazendo $s_3 = \delta(t_3)$ o resultado segue pela estabilidade de $\rightarrow$.

Geralmente para provar este caso nos livros textos, por exemplo em [4], assume-se que as regras de reescrita $l_i \rightarrow r_i$ são renomeadas de forma a obter que $Var(l_1) \cap Var(l_2) = \emptyset$. O que leva a assumir que a condição $Dom(\sigma_1) \cap Dom(\sigma_2) = \emptyset$ é satisfeita. E que consequentemente a substituição $\sigma_3 = \sigma_1 \cup \sigma_2$ é bem-definida e um unificador dos termos $l_1|_p$ e l_2. Daí conclui-se, após breve análise, que a divergência considerada é uma instância de um certo par crítico. No entanto, em nossa prova mecânica foi necessário formalizar um

lema auxiliar, tal como mostrado na Tabela 5.10.4, que certificasse que tal renomeamento existe. Observe que para a condição $Var(l_1) \cap Var(l_2) = \emptyset$ ser satisfeita é suficiente renomear apenas o lado esquerdo de uma das regras.

Tabela 5.10.4: Certificando que a propriedade $Var(l_1) \cap Var(l_2) = \emptyset$ é satisfeita

```
CP_lemma_aux1a: LEMMA
 FORALL E, (e1 | member(e1, E)), (e2 | member(e2, E)), (p: position):
 ( positionsOF(lhs(e1))(p)                                              &
   NOT vars?(subtermOF(lhs(e1), p))                                     &
   ext(sg1)(subtermOF(lhs(e1), p)) = ext(sg2)(lhs(e2)) )
        =>
          EXISTS alpha, rho:
             disjoint?(Vars(lhs(e1)), Vars(ext(rho)(lhs(e2))))          &
             ext(sg1)(subtermOF(lhs(e1), p)) = ext(comp(alpha, rho))(lhs(e2))
```

Caso 2b: $p = q_1 q_2$ tal que q_1 é uma posição de variável de l_1, e $\sigma_2(l_2) = \sigma_1(l_1|_{q_1})|_{q_2}$.

Este caso em muitos livros texto é apresentado pictoricamente sem muitos detalhes. No entanto, este caso é de difícil análise, pois o subtermo $l_1|_{q_1}$ pode ocorrer repedidamente em ambos os termos l_1 e r_1, e sua prova requer o auxílio do seguinte lema, cuja prova pode ser encontrada em [34].

Lema 5.10.3: Sejam $\rightarrow$ uma relação compatível, x uma variável fixa, e σ_1 e σ_2 substituições tais que:

$$\begin{array}{lcl} \sigma_1(x) & \rightarrow & \sigma_2(x) \\ \sigma_1(y) & = & \sigma_2(y) \text{ para todo } y \neq x. \end{array}$$

Seja t um termo arbitrário, e sejam $p_1, \cdots, p_n \in Pos(t)$ todas as ocorrências, distintas, de x em t. Definindo, $t_0 = \sigma_1(t)$ e $t_i = t_{i-1}[p_i \leftarrow \sigma_2(x)]$, para $1 \leq i \leq n$, temos que $t_i \rightarrow^{n-i} \sigma_2(t)$, para $0 \leq i \leq n$. Em particular, $\sigma_1(t) \rightarrow^n \sigma_2(t)$.

Para formalizar o lema anterior foi necessário a especificação de dois construtores auxiliares, como sugere o enunciado, chamados `replace_pos` e `RSigma` (Ver Tabela 5.10.5). Ressaltamos que, em geral, este caso foi o mais difícil de formalizar e exigiu a formalização de 13 lemas auxiliares que podem ser encontrados na *subteoria* `critical_pairs_aux`.

Tabela 5.10.5: Construtor `replace_pos`

```
replace_pos(t, s, (fssp:SPP(s)) ): RECURSIVE term =
   IF length(fssp) = 0
   THEN  s
   ELSE  replace_pos(t,replaceTerm(t, s, fssp(0)), rest(fssp))
   ENDIF
 MEASURE length(fssp)

 RSigma(R, sg1, sg2, x): bool = FORALL (y: (V)): IF y /= x
                                                 THEN sg1(y) = sg2(y)
                                                 ELSE R(sg1(x), sg2(x))
                                                 ENDIF
```

O construtor `replace_pos` possui três argumentos, e age trocando recursivamente a seqüência de posições do segundo argumento, pré-determinada no terceiro argumento, pelo primeiro argumento. Já o construtor `RSigma(R, sg1, sg2, x)` nos diz que, exceto para `x`, `sg1` e `sg2` possuem as mesmas imagens.

De posse dos construtores `replace_pos` e `RSigma` e suas propriedades, especificamos e formalizamos uma versão do Lema 5.10.3 tal como mostra a Tabela 5.10.6.

Tabela 5.10.6: Posição de Variável

```
CP_lemma_aux2: LEMMA
  FORALL R, t, x, sg1, sg2:
   LET Posv = Pos_var(t, x), seqv = set2seq(Posv) IN
    comp_cont?(R) & RSigma(R, sg1, sg2, x)
       =>
          (FORALL (i: below[length(seqv)]):
              RTC(R)(replace_pos(ext(sg2)(x),ext(sg1)(t), #(seqv(i))),ext(sg2)(t)))
                                  &
                RTC(R)(ext(sg1)(t), ext(sg2)(t))
```

Na Tabela 5.10.6, `Pos_var(t,x)` é o conjunto das posições do termo `t` onde ocorre a variável `x`, tal como especificado na Seção 5.4, e o construtor `#(*)` representa a seqüência unitária contendo o elemento `*`. Além disso, foi necessário usar o construtor `set2seq`, que pode ser encontrado na *sub-teoria* `finite_sequences_extras`, o qual transforma um conjunto finito em uma seqüência finita de seus elementos, em nosso caso este construtor

foi utilizado para converter um conjunto finito de posições de um dado termo em uma seqüência finita de posições. Esta conversão foi extremamente útil, pois possibilitou provas indutivas no comprimento de uma seqüência. ■

A formalização do Teorema dos Pares Críticos de Knuth-Bendix em PVS é extensa e exigiu a formalização de 16 lemas auxiliares, além daqueles referentes à propriedades sobre substituição, troca de subtermos, seqüências, e etc.. Vale lembrar que dos 16 lemas auxiliares formalizados, 14 deles são referentes ao caso "posição de variável", os quais podem ser encontrados na *sub-teoria* `critical_pairs_aux`. Além disso, para completar a formalização deste teorema foram usados aproximadamente 933 comandos (regras) de prova, sem contar os comandos usados para formalizar os 16 lemas auxiliares.

Conclusão e Trabalhos Futuros

Apresentou-se formalizações para as teorias de Sistemas Abstratos de Redução e Sistemas de Reescrita de Termos, chamadas `ars` e `trs`, desenvolvidas no assistente de prova PVS. Até onde sabemos, não existem outras formalizações dessas teorias em PVS. As *teorias* `ars` e `trs` fornecem uma base sólida que permite o desenvolvimento de conceitos e resultados da teoria de reescrita em geral, além daqueles especificados e formalizados nestas *teorias*. Evidência clara desta afirmação é fornecida pela formalização de resultados elaborados e não triviais das teorias de ARS e TRS, como por exemplo:

- Lema de Newman;
- Lema de Yokouchi;
- Teorema dos Pares Críticos de Knuth-Bendix.

Note também que estas formalizações são evidências da boa integração entre diferentes *sub-teorias* tanto de `ars` quanto de `trs`, as quais foram desenvolvidas separadamente e combinadas para formalizar tais resultados. Além da generalidade, destacou-se da formalização das *teorias* `ars` e `trs` o seguinte:

- o uso de uma linguagem de especificação de ordem superior, o que é natural para expressar propriedades e conceitos de objetos de segunda ordem como as relações de redução; e

- o alto nível de abstração, que permite observar em forma quasi-geométrica as propriedades destas teorias e, em particular, da teoria de ARS na qual analiticamente a associação de propriedades das relações com diagramas é a metodologia padrão.

Como mencionado na Seção 1.3 sobre trabalhos relacionados, em [61] é apresentada uma formalização em linguagem de primeira ordem do Teorema dos Pares Críticos de Knuth-Bendix, que é reportada como a única formalização existente na época. Não conhecemos outras formalizações completas deste teorema e assim acreditamos que a formalização apresentada neste trabalho deste teorema seja a primeira realizada utilizando uma linguagem de especificação de ordem superior como PVS.

Nas formalizações dos Lemas de Newman e Yokouchi apresentadas na Seção 4.6 fica clara a utilização da associação com diagramas nas provas. Dentre outras formalizações que foram baseadas em diagramas e não detalhadas neste trabalho citamos: o Lema da Comutação e o Lema da União Comutativa.

Tabela 5.10.7: Análise Quantitativa das *Teorias* `ars` e `trs`.

Teoria	L. Especificação	L. Prova	Teoremas	TCCs	T. Especificação	T. Prova
`ars`	791	8384	60	5	48K	640K
`trs`	1980	42105	166	119	108K	2.8M
Total	2745	50489	226	124	156K	3.4M

A tabela 5.10.7 apresenta algumas informações quantitativas sobre as *teorias* `ars` e `trs`. A primeira coluna na tabela 5.10.7 contém os nomes das *teorias*. A segunda coluna e a terceira coluna mostram o número de linhas (incluindo comentários e linhas em branco) que foram gastas para especificar e provar todos os conceitos e resultados das respectivas *teorias*. A quarta coluna mostra o número de TCCs gerados, ou seja, as obrigações de prova geradas automaticamente pelo PVS. Dentre o total de TCCs apresentados, 62 TCCs foram provados manualmente e dentre estes, apenas 3 TCCs são relacionados com a *teoria* `ars`, especificamente com a *sub-teoria* `relations_closure`, e 21 TCCs são relacionados com o Teorema dos Pares Críticos de Knuth-Bendix. As duas últimas colunas mostram as *teorias* `ars` e `trs` em tamanhos.

As evidências mostram que as *teorias* `ars` e `trs` formam uma base de conhecimentos formalizados que podem ser continuamente estendidos e desenvolvidos. Por exemplo, em geral, outros resultados interessantes da Teoria de Reescrita podem ser formalizados baseados na *teoria* `trs`. Por exemplo, a propriedade de confluência dos *Sistemas Ortogonais* pode ser demonstrada reutilizando as mesmas técnicas usadas para a formalização do Teorema dos Pares Críticos de Knuth-Bendix. Em particular, a restrição de ser *linear à esquerda* é um caso particular do **Caso 2b** da prova deste teorema apresentado na Seção 5.10.

Como mencionamos na Seção 5.7, até o momento a *teoria* `trs` não inclui uma *subteoria* para tratar da existência e unicidade de *unificadores mais gerais* e tratamos desta existência e unicidade como um axioma. No entanto, esforços já estão sendo depositados neste sentido. Vale ressaltar que verificações da correção de algoritmos de unificação já foram desenvolvidos em outros assistentes de prova, por exemplo, em LCF por Paulson [55], em Boyer-Moore por Kaufman [38], e em Coq por Rouyer [60].

Já é um fato que terminação é uma importante propriedade de TRS. Por exemplo, a confluência de sistemas de reescrita se torna decidível quando o sistema em questão é terminante. Mas infelizmente, o *problema de terminação* para TRS é, em geral, indecidível. No entanto, para sistemas de reescrita particulares existem vários critérios muito bem estabelecidos, ver por exemplo [4] e [6], que facilitam verificar a terminação de tais sistemas.

Como vimos na Seção 4.5, a idéia básica para estabelecer critérios para terminação é fornecer uma ordem bem-fundada (noeteriana) para o conjunto dos termos. Em outras palavras, o principal problema é fornecer uma ordem apropriada que garanta a terminação, e é isto o que fazem os seguintes critérios:

- *Método de Interpretações:* Este critério em vez de considerar uma ordem sobre o conjunto dos termos, considera sua interpretação em uma Σ-álgebra a qual é equipada com uma ordem bem-fundada. Por exemplo, as chamadas *ordens polinomiais* para as quais cada símbolo de função é interpretado como um polinômio sobre os números naturais de acordo com a sua aridade.

- *Ordens de Simplificação:* Este critério se baseia em ordens estritas compatíveis com operações e fechadas para substituições (Ver Seções 5.6 e 5.9) de forma que todo termo é "maior" do que seus subtermos próprios. Dessa forma obtemos ordens bem-fundadas, como podemos ver em [4].

Devido ao supracitado, pretendemos focalizar nossa atenção na formalização de critérios para garantir terminação, como por exemplo, os citados acima. Devemos ressaltar que critérios para terminação já foram formalizados como mostram as bibliotecas CoLor [7] e Coccinelle [13].

Apêndice A

Sintaxe da Linguagem de Especificação do PVS

Basicamente uma linguagem de especificação[1] é um meio de expressar "*o que*" é computado em vez de "*como*" é computado. Em outras palavras, uma linguagem de especificação é uma lógica dentro da qual o comportamento de sistemas computacionais podem ser formalizados, ou seja, o comportamento de tais sistemas podem ser simulados. Todavia, usamos as especificações para declarar e provar propriedades de sistemas com ajuda mecânica.

O PVS consiste de uma *linguagem de especificação* [53] projetada para admitir especificações sucintas, legíveis, e para efetivar construções de provas em vez de execuções eficientes. A linguagem de especificação do PVS é construída sobre uma lógica de ordem superior, ou seja, significa que as variáveis podem variar sobre relações, relações de relações, e assim por diante.

As especificações em PVS são organizadas como uma coleção de *teorias*, e quando uma *teoria* é introduzida no sistema cria-se um arquivo cuja extensão é `pvs` . Uma *teoria* simples, sem parâmetros (Veja Seção 3.4), possui a estrutura tal como apresentada na Tabela A.1.

O corpo de uma *teoria* é essencialmente composto de *declarações*, as quais são usadas para introduzir tipos, constantes (incluindo funções), variáveis, axiomas e fórmulas, e IMPORTINGs, com os quais importam-se outras *teorias*.

[1]Todo este Apêndice é baseado nos manuais [54] e [53] do próprio PVS.

Tabela A.1: Estrutura de uma *teoria* sem parâmetros.

```
exemplo : THEORY
  BEGIN

       Corpo da Teoria

  END exemplo
```

- *Declarações de Tipo* são usadas para introduzir novos nomes de tipo. Por exemplo, `T: TYPE+` introduz `T` como um tipo "não interpretado" e não vazio. Vale ressaltar que o uso de tipos "não interpretado" proporciona um maior nível de abstração em uma especificação em PVS. O PVS também permite várias outras espécies de declarações de tipo tais como declaração de tipos "interpretados", declaração de tipos "enumerados", etc.

- *Declarações de Constantes* são usadas para introduzir novas constantes, e em PVS, os termos constantes referem-se a funções e relações, assim como as constantes no sentido usual (0-ária). Por exemplo, `n: posnat` introduz `n` como uma constante natural positiva não-interpretada. O PVS possui uma variedade de expressões construtoras tais como operadores lógicos e aritméticos, funções aplicações, lambda abstrações, expressões `IF-THEN-ELSE`, etc.

- *Declarações de Fórmulas* são usadas para introduzir *axiomas*, *suposições*, *obrigações* e *teoremas*, e são as únicas declarações que permitem o uso de variáveis livres. Os axiomas, suposições, e obrigações são introduzidos usando, respectivamente, as palavras chave `AXIOM`, `ASSUMPTION` e `OBLIGATION`. Já os teoremas podem ser introduzidos por meio das seguintes e equivalentes palavras chaves `CONJECTURE`, `CLAIM`, `LEMMA`, ou `THEOREM`. Estes correspondem à propriedades que desejamos provar sobre a nossa especificação.

A linguagem de especificação do PVS também suporta modularidade por meio de *teorias* parametrizadas, como vimos na Seção 3.4. Vale ressaltar que *teorias* parametrizadas são muito convenientes uma vez que o uso do parâmetro permite especificações mais gerais. Vejamos um exemplo:

Tabela A.2: Estrutura de uma *teoria* parametrizada.

```
parameter[T : TYPE] : THEORY
  BEGIN
      x, y: VAR T
    f(x,y): T = x*x - y*y
      plus: LEMMA f(x,y) + f(y,x) = 0
     minus: LEMMA f(x,y) - f(y,x) = 2*f(x,y)
  END parameter
```

No exemplo apresentado na Tabela A.2, T é um parâmetro para a *teoria* `parameter` e é tratado como um tipo "fixo" não interpretado, `x` e `y` são variáveis de tipo `T`, `f` é uma função (constante) de tipo `[[T,T] -> T]`, e `plus` e `minus` são propriedades que desejamos provar. Observe que devido ao fato da *teoria* estar parametrizada por `T`, quando invocarmos a *teoria* `parameter` por uma outra *teoria*, `T` deve ser instanciado. Por exemplo, a *teoria* `parameter` aplicada aos números reais deve ser invocada como `parameter[real]`.

A *teoria* para Grupos apresentada na Tabela A.3 ilustra vários dos conceitos que apresentamos anteriormente. A *teoria* `groups` possui 4 parâmetros: um tipo `G`, um elemento identidade `e` de `G`, e duas operações `o` e `inv` sobre elementos de `G`. Observe o uso do parâmetro `G` nos outros parâmetros da *teoria*.

Tabela A.3: Uma *teoria* para grupos

```
groups [  G : TYPE,
          e : G,
          o : [G,G->G],
        inv : [G->G]    ] : THEORY
  BEGIN
    ASSUMING
      a, b, c: VAR G
      associativity : ASSUMPTION a o (b o c) = (a o b) o c
      unit : ASSUMPTION e o a = a AND a o e = a
      inverse : ASSUMPTION inv(a) o a = e AND a o inv(a) = e
    ENDASSUMING
    left_cancellation: THEOREM a o b = a o c IMPLIES b = c
    right_cancellation: THEOREM b o a = c o a IMPLIES b = c
  END groups
```

Apêndice B

O Assistente de Prova do PVS

Provavelmente o PVS é mais conhecido pelo poder e facilidade em usar seu *assistente de prova*[1], o qual é usado interativamente para construir a prova de uma conjectura (*objetivo*). O assistente de prova do PVS foi desenvolvido com base na Teoria da Prova, tal como apresentado no Capítulo 2. Sendo assim, os objetivos em PVS são sequentes da forma $\Sigma \vdash \Lambda$, onde Σ e Λ são sequências finitas de fórmulas, com a semântica usual de Gentzen, tal como apresentado na Seção 2.1.2.

Quando o objetivo é mostrado para o usuário, os antecedentes (membros de Σ) são numerados com números negativos e os consequentes (membros de Λ) são numerados com números positivos, como mostrado na Tabela B.1, e estes números são usados para endereçar os componentes do objetivo.

Tabela B.1: Antecedentes e consequentes

[-1] A_1
[-2] A_2
$\vdots$
|-----
[1] C_1
[2] C_2
$\vdots$

Quando o assistente de prova é invocado sobre um teorema a ser provado, a árvore de prova inicia com um nó raiz (sequente) que não possui antecedentes e tendo como

[1]Todo este Apêndice é baseado nos manuais [54] e [66] do próprio PVS.

consequente somente o teorema a ser provado. Também durante toda a prova, a atenção do assistente de prova é voltada para algum objetivo (sequente) que está mais à esquerda na árvore da prova corrente e o assistente mostra este sequente enquanto espera uma ação (comando de prova) do usuário. Uma vez fornecido um comando de prova, este pode causar o reconhecimento do objetivo como `TRUE`, ou seja, sua árvore de prova está terminada, ou pode gerar mais sub-árvores (subobjetivos).

Para ilustrar estes conceitos vejamos o exemplo apresentado na Tabela B.2, que é usado para introduzir o PVS em [54]. Assim, suponha que a *teoria* `sum.pvs` contenha:

Tabela B.2: *Teoria* `sum`

```
sum: THEORY
  BEGIN

    n: VAR nat

    sum(n): RECURSIVE nat =
      (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
      MEASURE n

    closed_form: THEOREM sum(n) = (n * (n + 1)) / 2
  END sum
```

A *teoria* `sum.pvs` contém:

1. uma variável `n` que é declarada para ter o tipo (pre-definido) `nat`;
2. uma função `sum` que é definida recursivamente. Note que a *boa-formação* da recursão é explicitamente justificada fornecendo a medida `MEASURE n`;
3. um teorema chamado `closed_form` a ser provado.

Quando invocamos o PVS sobre o arquivo `sum.pvs` uma janela do Emacs contendo seu conteúdo é aberta. Para provar o teorema, o cursor deve ser posicionado sobre a linha que contém o teorema `closed_form`, e então `ALT-x prove` deve ser digitado no mini-buffer do Emacs. Assim, o typechecking da *teoria* contendo o teorema a ser provado é iniciado, e uma vez terminado, o sequente contendo o teorema a ser provado é mostrado com o

prompt `Rule?` onde se deve entrar com um comando de prova como mostrado a seguir. O typechecking pode gerar os chamados *TCCs* (type correctness conditions) analisando o uso dos subtipos predicados em expressões. Uma expressão em PVS não é considerada totalmente checada a menos que todos os TCCs tenham sido provados corretos.

```
closed_form :

|-------
{1} FORALL (n: nat): sum(n) = n * (n + 1) / 2

Rule?
```

Como é sabido este teorema pode ser provado usando indução natural sobre `n`. Então entrando com o comando de prova `(induct "n")` no prompt `Rule?` obtemos dois subobjetivos (`closed_form.1` e `closed_form.2`):

```
closed_form.1 :

  |-------
{1}   sum(0) = (0 * (0 + 1)) / 2

Rule?
```

O primeiro subobjetivo que é apresentado para o usuário refere-se à base de indução (caso `n=0`). Este subobjetivo é provado usando o comando de prova `(grind)`. Uma vez provado o primeiro subobjetivo, o segundo é apresentado ao usuário e refere-se ao passo indutivo.

```
closed_form.2 :

  |-------
{1}   FORALL j:
        sum(j) = (j * (j + 1)) / 2 IMPLIES
         sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2
Rule?
```

Da mesma forma, este subobjetivo é provado usando o comando `(grind)` e, portanto, o teorema esta provado. No entanto, a prova ainda não pode ser considerada *completa*. Vejamos o porque!

Quando fizemos o typechecking da *teoria* `sum.pvs` o mini-buffer do Emacs relatou que existem dois TCCs ainda não provados. Se digitamos `ALT-x spt` podemos visualizar os status das provas da *teoria* (spt) em questão, o qual é apresentado na Tabela B.3.

Tabela B.3: Status das provas da *teoria* `sum`: *Teoria* incompleta

```
Proof summary for theory sum
  sum_TCC1.............................unfinished          [shostak]( n/a s)
  sum_TCC2.............................unfinished          [shostak]( n/a s)
  closed_form..........................proved - incomplete [shostak](1.62 s)
  Theory totals: 3 formulas, 3 attempted, 1 succeeded (1.62 s)
```

O status das fórmulas apresentado pelo PVS pode assumir uma das quatro possibilidades: `untried` significa que nenhuma prova foi tentada, `proved` significa que a prova foi completada, `unchecked` significa que a prova foi completada, mas houve uma modificação na especificação desde a última tentativa, e `unfinished` significa que uma prova foi tentada, mas ainda não completada. Provas rotuladas como `proved` podem ser `complete` ou `incomplete`. Será considerada `complete` *quando todas as fórmulas (incluindo os TCCs) dos quais ela depende já tenham sido provados.*

Os TCCs que aparecem no relatório apresentado na Tabela B.3 podem ser visualizados digitando `ALT-x show-tccs` e são mostrados na Tabela B.4.

Tabela B.4: TCCs gerados para a *teoria* `sum.pvs`

```
% Subtype TCC generated (at line 7, column 36) for  n - 1
    % expected type  nat
  % unfinished
sum_TCC1: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0;

% Termination TCC generated (at line 7, column 32) for  sum(n - 1)
  % unfinished
sum_TCC2: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n;
```

O primeiro TCC (`sum_TCC1`) pede para ser provado que o argumento `n` é sempre não-negativo. Isto porque na lógica do PVS o tipo `nat` é um subtipo dos inteiros e, portanto, `0 - 1` = `-1`. Mas observe que o TCC inclui a condição `NOT n = 0` e, consequentemente, pode ser provado.

O segundo TCC (`sum_TCC2`) é necessário para garantir que a função `sum` é total, ou seja, termina. Apesar de usarmos uma medida (`MEASURE`) para mostrar que as definições recursivas são totais, se faz necessário garantir que a medida decresce a cada chamado recursivo.

Mas nesse caso não devemos nos preocupar, pois estes TCCs são triviais e podem ser provados automaticamente digitando (no mini-buffer do Emacs) `ALT-x tcp` o qual tenta provar todos os TCCs gerados. Feito isso, os dois TCCs são descarregados, e novamente usando `ALT-x spt` podemos visualizar o status das provas da *teoria* `sum` e verificar que todas estão completas, como mostrado na Tabela B.5.

Tabela B.5: Status das provas da *teoria* `sum`: *Teoria* completa

```
Proof summary for theory sum
   sum_TCC1.............................proved - complete   [shostak](0.06 s)
   sum_TCC2.............................proved - complete   [shostak](0.02 s)
   closed_form..........................proved - complete   [shostak](1.62 s)
   Theory totals: 3 formulas, 3 attempted, 3 succeeded (1.70 s)
```

Como já comentamos antes, o PVS fornece uma interface com o Emacs para construir as provas interativamente e salva automaticamente as provas num arquivo com extensão `.prf`. O PVS também fornece uma interface com o Tcl/TK para mostrar graficamente as árvores de prova. Dessa forma, depois de completar uma prova digitando `ALT-x x-show-proof` uma janela é aberta contendo a árvore de prova. Por exemplo, para o teorema `closed_form` a árvore é como apresentada na Figura B.1:

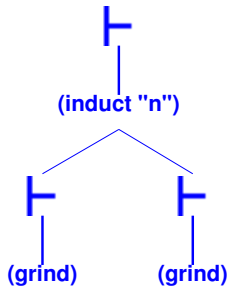


Figura B.1: Árvore de prova do teorema `closed_form`

Apêndice C

Comandos de Prova no PVS

Os comandos de prova[1] no PVS são *regras* ou *estratégias*. Uma *regra* executa um passo atômico no assistente de prova e é um comando que pode ser invocado pelo nome e, se apropriado, aplicado a argumentos. Agora uma *estratégia* é um comando que combina aplicações de uma ou mais regras, e outras estratégias. Dessa forma uma regra é na verdade uma estratégia degenerada.

Um comando de prova quando aplicado a um objetivo (sequente) ele produz uma das seguintes:

1. Sucesso na prova do objetivo;
2. Gera um ou mais subobjetivos;
3. Não produz efeito nenhum;
4. Sinal de falha;
5. Adia a construção da prova do objetivo corrente, transferindo o foco para o próximo subobjetivo remanescente.

A seguir apresentamos algumas das classes de comandos de prova implementados no PVS e descrevemos alguns exemplos de regras. Para mais comandos e maiores detalhes ver [66].

[1]Todo este Apêndice é baseado no manual [66] do próprio PVS.

1. Controle:

   (a) `postpone`: Deixa o objetivo corrente pendente e vai para o próximo objetivo, se existir.

   Exemplo: `(postpone)`

   (b) `quit`: Sai do assistente de prova questionando se queremos salvar a prova corrente.

   Exemplo: `(quit)`

2. Regras estruturais:

   (a) `copy`: Faz uma cópia de uma fórmula.

   Exemplo: `(copy -4)` insere uma cópia da fórmula numerada com `-4` como o primeiro antecedente do corrente sequente, ou seja, numerada com `-1`.

   (b) `hide`: Torna fórmulas invisíveis.

   Exemplo: `(hide (-1 -4 2 3))` fornece um novo sequente como resultado de tornar invisíveis as fórmulas `-1`, `-4`, `2` e `3` no corrente sequente.

   (c) `hide-all-but`: Torna fórmulas invisíveis.

   Exemplo:`(hide-all-but (-2 3)` fornece um novo sequente como resultado de tornar invisíveis todas as fórmulas do corrente sequente exceto `-2` e `3`.

   (d) `reveal`: Torna visíveis fórmulas invisíveis. Para visualizar as fórmulas (em forma de sequente) que foram tornadas invisíveis use o comando `ALT-x show-hidden` no mini-buffer do Emacs.

   Exemplo: `(reveal (-3 4))` torna visíveis as fórmulas `-3` e `4` mostradas no sequente obtido com o comando `ALT-x show-hidden`.

3. Regras proposicionais:

   (a) `flatten`: Aplica simplificação disjuntiva.

   Exemplo: `(flatten)` aplica simplificação disjuntiva (conjunção no antecedente e disjunção no consequente) em todas as fórmulas do corrente sequente.

(b) `case`: Gera dois subobjetivos onde a expressão boleana dada é assumida ser `TRUE` em um, e `FALSE` em outro.

Exemplo: `(case "x > 0")` gera a partir do corrente sequente ($\Gamma \vdash \Delta$) os seguintes subobjetivos: `x > 0`, $\Gamma \vdash \Delta$ e $\Gamma \vdash$ `x > 0`, Δ

(c) `prop`: Faz simplificação proposicional

Exemplo: (prop)

4. Regras para tratamento de quantificadores:

(a) `inst`: Faz a instanciação de um quantificador universal no antecedente ou um quantificador existêncial no consequente.

Exemplo: `(inst -3 "a")` instância o primeiro quantificador universal no antecedente `-3`, com exatamente uma variável ligada, com `a`.

(b) `skolem`: Introduz constantes de skolem para quantificadores universais no consequente (provando o objetivo sem perda de generalidade) ou constantes para quantificadores existências no antecedente quando é sabido que existem. Em outras palavras, `skolem` introduz novos nomes para constantes, isto é, para a variável `x` ele introduzirá `x!1`, `x!2`, ... quando aplicado repetidamente.

Exemplo: `(skolem * "a1" "a2" "a3")`, a primeira fórmula, conveniente, do sequente da forma ($\forall/\exists\ x_1, x_2, x_3 : A$) é repassada por A[`a1`/x_1, `a2`/x_2, `a3`/x_3].

(c) `skosimp*`: Aplica repetidamente `skolem` e `flatten`

Exemplo: `(skosimp*)`

5. Regras de igualdade:

(a) `replace`: Reescreve usando igualdade sintática.

Exemplo: `(replace -1)`, se a fórmula `-1`, no corrente sequente, é da forma $l = r$, então este comando gera um novo sequente no qual todas as ocorrências de um termo sintaticamente equivalente à l são trocadas por r.

(b) `case-repalce`: Introduz e repassa igualdades.

Exemplo: `(case-replace "a = b")` troca `a` por `b` no corrente sequente e gera um segundo subobjetivo onde somos obrigados a provar que `a = b`.

6. Regras para definições e lemas:

(a) `expand`: Expande definições.

Exemplo: `(expand "sum"1)` expande todas as ocorrências da expressão definida como `sum` na fórmula 1.

(b) `lemma`: Introduz instâncias de lemas, teoremas, axiomas e etc.

Exemplo: `(lemma "nome do teorema")` adiciona no corrente sequente a fórmula correspondente ao conteúdo do teorema `nome do teorema`.

(c) `rewrite`: Procura matching e reescreve com lemas, teoremas e etc.

Exemplo: `(rewrite "teorema")` encontra, se existir, e reescreve uma instância do teorema "teorema"no corrente sequente.

7. Regras para simplificação:

(a) `assert`: Simplifica usando procedimentos de decisão.

Exemplo: `(assert)`

(b) `grind`: Usa reescrita e aplica simplificação repetidamente.

Exemplo: `(grind)`

Tipicamente os comandos possuem argumentos. Se nenhum argumento é fornecido, então a versão *força bruta* do comando é usada, o que, em geral, é o que os usuários tentam primeiro. No entanto, quando fornecidos argumentos a aplicação do comando se torna mais controlável e consequentemente o objetivo a ser provado. Por exemplo, sem fornecer argumentos para a regra `(flatten)`, este aplica simplificação disjuntiva em ambos antecedentes e consequentes. Agora, usando-o com argumentos `+` (`(flatten +)`), este aplica a simplificação apenas nos consequentes, e se usamos `-` (`(flatten -)`), este aplica a simplicação somente nos antecedentes. Além disso, se o usuário desejar, pode

aplicar a regra `(flatten)` somente à fórmulas específicas fornecendo como argumentos os números de tais fórmulas, por exemplo, `(flatten (-2 3 4))`.

Referências Bibliográficas

[1] Thorsten Altenkirch. A Formalization of the Strong Normalization Proof for System F in LEGO. In Marc Bezem and Jan Friso Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[2] Thorsten Altenkirch. Proving Strong Normalization of CC by Modifying Realizability Semantics. In Henk P. Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 3–18. Springer-Verlag, 1994.

[3] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by Evaluation for Typed Lambda Calculus with Coproducts. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[4] Franz Baader and Tobias Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[5] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[6] Marc Bezem, Jan Willem Klop, and Roel de Vrijer, editors. *Term Rewriting Systems by TeReSe*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[7] Frédéric Blanqui, Solange Coupet-Grimal, William Delobel, Sb́astien Hinderer, and Adam Koprowski. CoLoR, a Coq Library on Rewriting and termination. In *8th International Workshop on Termination (WST '06)*, 2006.

[8] Mirna Bognar. A Survey of Abstract Rewriting (draft), June 1995. Disponível: `http://citeseer.ist.psu.edu/431795.html` (Visitada em 08/09/2008).

[9] Robert S. Boyer and J. Strother Moore. *A computational logic handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[10] Barras Bruno. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[11] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[12] Robert L. Constable, Stuart F. Allen, H. M. Bromley, Walter Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, , and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[13] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In Boris Konev and Frank Wolter, editors, *6th International Symposium on Frontiers of Combining Systems (FroCos 07)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Liverpool,UK, september 2007. Springer-Verlag.

[14] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Disponível: `http://www.csl.sri.com/wift-tutorial.html` (Visitada em 08/09/2008).

[15] Haskell Brookes Curry. Functionality in Combinatory Logic. *Proc. Nat. Acad Science USA*, 20:584–590, 1934.

[16] Haskell Brookes Curry and Robert Feys. *Combinatory Logic, Volume I.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, third edition, 1974.

[17] Heinz-Dieter Ebbinghaus, Jorg Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, second edition, 1994.

[18] Ford, Jonathan M. and Mason, Ian A. Operational Techniques in PVS – A Preliminary Evaluation. In *Proceedings of the Australasian Theory Symposium, CATS'01*, 2001.

[19] André Luiz Galdino and Mauricio Ayala-Rincón. A Theory for Abstract Rewriting Systems in PVS. In *XXXIII Conferencia Latinoamericana de Informática - CLEI'07*, 2007. Versão estendida convidada como um dos dez melhores trabalhos para publicação na Revista Eletrônica do CLEI. Disponível: `www.mat.unb.br/~ayala/publications.html` (Visitada em 08/09/2008).

[20] André Luiz Galdino and Mauricio Ayala-Rincón. A PVS *Theory* for Term Rewriting Systems. In Elaine Pimentel and Mario Benevides, editors, *Proceedings of the Third Workshop on Logical and Semantic Frameworks, with Applications - LSFA 2008*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008. Aceito. Disponível: `www.mat.unb.br/~ayala/publications.html` (Visitada em 08/09/2008).

[21] André Luiz Galdino and Mauricio Ayala-Rincón. Verification of Newman's and Yokouchi's Lemmas in PVS. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Local Proceedings of Logic and Theory of Algorithms, Fourth Conference on Computability in Europe - CiE 2008*, pages 137–146. University of Athens, 2008. Disponível: `www.mat.unb.br/~ayala/publications.html` (Visitada em 08/09/2008).

[22] André Luiz Galdino, César A. Muñoz, and Mauricio Ayala-Rincón. Formal Verification of an Optimal Air Traffic Conflict Resolution and Recovery Algorithm. In D. Leivant and R. de Queiroz, editors, *Proceedings of the 14th Workshop on Logic, Language, Information and Computation - WoLLIC 2007*, volume 4576 of *Lecture Notes in Computer Science*, pages 177–188, Rio de Janeiro, Brazil, July 2007. Springer-Verlag.

[23] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschirift*, pages 172–210 and 405–431, 1934/5. English translation: "Investigations into Logical Deduction" in The Collected Works of Gehard Gentzen, ed. M.E.Szabo, North-Holland Pub Co.,1969.

[24] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, New York, NY, USA, 1989.

[25] Michael Gordon, Arthur J. Milner, and Christopher P. Wadsworth. Edinburgh LCF: A Mechanized Logic of Computation. In *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*, pages 280–291. Springer-Verlag, 1979.

[26] Keith Hanna and Neil Daeche. The Veritas Design Logic: A User's View. In V. Stavridon, T.F. Melham, and R.T. Boute, editors, *International Conference on Theorem Provers in Circuit Design*, volume A-10 of *IFIP Transactions*, pages 301–310, Nijmegen, June 1992. North-Holland.

[27] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[28] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie-Claude Gaudel and Jim Woodcock, editors, *Third International Symposium of Formal Methods Europe (FME'96)*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer-Verlag, 1996.

[29] J. Roger Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997.

[30] Jozef Hooman. Correctness of Real Time Systems by Construction. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 19–40, London, UK, 1994. Springer-Verlag.

[31] Jozef Hooman. Verifying Part of the ACCESS.bus Protocol Using PVS. In P.S. Thiagarajan, editor, *15th Conference on the Foundations of Software Technology and*

Theoretical Computer Science, volume 1026 of *Lecture Notes in Computer Science*, pages 96–110, Bangalore, India, 1995. Springer-Verlag.

[32] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[33] William A. Howard. The Formulae-As-Types Notion Of Construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, N.Y., 1980.

[34] Gérard Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, 1980.

[35] Gérard Huet. Residual Theory in λ-calculus: A Formal development. *Jornal of Functional Programming*, 4(3):371–394, 1994.

[36] Graham Hutton. An Introduction to HOL, a Theorem Proving Environment for Higher Order Logic. *Journal of Functional Programming*, 4(4):557–559, October 1994.

[37] Stanislaw Jaskowski. On the Rules of Suppositions in Formal Logic. *Studia Logica*, 1:5–32, 1934.

[38] Matt Kaufmann. Generalization in the Presence of Free Variables: A Mechanically-Checked Proof for one Algorithm. *Journal of Automated Reasoning*, 7(1):109–158, 1991.

[39] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[40] Florent Kirchner and César A. Muñoz. PVS#: Streamlined Tacticals for PVS. *Electronic Notes in Theoretical Computer Science*, 174(11):47–58, 2007.

[41] Stephen Cole Kleene. *Introduction to Metamathematics*. Number 1 in Bibliotheca mathematica. North-Holland, Amsterdam, 1952. Revised edition, Wolters-Noordhoff, 1971.

[42] Jan Willem Klop. Term rewriting systems. In *Handbook of logic in computer science (vol. 2): background: computational structures*, pages 1–116. Oxford University Press, Inc., New York, NY, USA, 1992.

[43] Donald E. Knuth and P. B. Bendix. Simple Word Problems in Universal Algebra. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[44] Adam Koprowski. A Formalization of the Simply Typed Lambda in Coq (draft), 2006. Disponível: `http://www.win.tue.nl/~akoprows/publications.html` (Visitada em 08/09/2008).

[45] Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Computer Science Dept., Univ. of Edinburgh, 1992.

[46] James McKinna and Robert Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, 23(3-4):373–409, 1999.

[47] Maxwell Herman Alexander Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[48] Tobias Nipkow. More Church-Rosser Proofs. *Journal of Automated Reasoning*, 26(1):51–66, 2001.

[49] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[50] Sam Owre and Natarajan Shankar. Abstract Datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997; Also available as NASA Contractor Report CR-97-206264. Disponível: `http://pvs.csl.sri.com/` (Visitada em 08/09/2008).

[51] Sam Owre and Natarajan Shankar. The Formal semantics of PVS. Technical report, SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997. Disponível: `http://pvs.csl.sri.com/` (Visitada em 08/09/2008).

[52] Sam Owre and Natarajan Shankar. The PVS Prelude Library. Technical report, SRI-CSL-03-01, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2003. Disponível: `http://pvs.csl.sri.com/` (Visitada em 08/09/2008).

[53] Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. Disponível: `http://pvs.csl.sri.com/` (Visitada em 08/09/2008).

[54] Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. Disponível: `http://pvs.csl.sri.com/` (Visitada em 08/09/2008).

[55] Lawrence C. Paulson. Verifying the Unification Algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, 1985.

[56] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the Association for Computing Machinery*, 28(2):233–264, April 1981.

[57] Frank Pfenning. A Proof of the Church-Rosser Theorem and its Representation in a Logical Framework, September 1992. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186. Disponível: http://www.cs.cmu.edu/∼fp/papers/cr92.ps (Visitada em 08/09/2008).

[58] NASA Langley PVS Libraries, July 2008. Disponível: `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html` (Visitada em 08/09/2008).

[59] Ole Rasmussen. The Church-Rosser Theorem in Isabelle: A Proof Porting Experiment. Technical Report 364, University of Cambridge, Computer Laboratory, March 1995.

[60] Joseph Rouyer. Développement de L'algorithm D'unification dans le Calcul des Constructions Avec Types Inductifs. Technical Report INRIA-RR - 1795, INRIA Lorraine, Le Chesnay, FRANCE (1980-200), 1992.

[61] José Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo, and Francisco-Jesús Martín-Mateos. Formalizing Rewriting in the ACL2 Theorem Prover. In *AISC'00: Revised Papers from the International Conference on Artificial Intelligence and Symbolic Computation*, volume 1930 of *Lecture Notes in Computer Science*, pages 92–106, London, UK, 2001. Springer-Verlag.

[62] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908. Reprinted in B. Russell, "Logic and Knowledge", London: Allen & Unwin, 1956, 59-102, and in J. van Heijenoort, "From Frege to Gödel", Cambridge, Mass.: Harvard University Press, 1967, 152-182.

[63] Amokrane Saïbi. Formalization of a lambda-Calculus with Explicit Substitutions in Coq. In *TYPES'94: Selected papers from the International Workshop on Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 183–202, London, UK, 1995. Springer-Verlag.

[64] Natarajan Shankar. A Mechanical Proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery*, 35:475–522, 1988.

[65] Natarajan Shankar. Verification of Real-Time Systems Using PVS. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, Elounda, Greece, 1993. Springer-Verlag.

[66] Natarajan Shankar, Sam Owre, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. Disponível: `http://pvs.csl.sri.com/` (Visitada em 08/09/2008).

[67] Morten Heine B. Sorensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 1. Elsevier Science, 2006.

[68] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Number 43 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, New York, NY, USA, 1996.

[69] Dirk van Dalen. *Logic and Struture*. Springer-Verlag, 1996.

[70] Vincent van Oostrom. Development Closed Critical Pairs. In *HOA'95: Selected Papers from the Second International Workshop on Higher-Order Algebra, Logic, and Term Rewriting*, volume 1074 of *Lecture Notes in Computer Science*, pages 185–200, London, UK, 1996. Springer-Verlag.

[71] Hirofumi Yokouchi. Church-Rosser theorem for a rewriting system on categorical combinators. *Theoretical Computer Science*, 65(3):271–290, 1989.