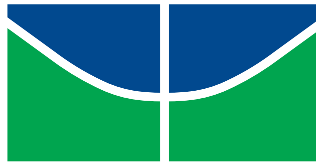


UNIVERSIDADE DE BRASÍLIA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE MATEMÁTICA

Formalização da Prova do Teorema de Existência de Unificadores Mais Gerais em Teorias de Primeira-Ordem

Por

Andréia Borges Avelar

Brasília
2009

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

UNIVERSIDADE DE BRASÍLIA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE MATEMÁTICA

Formalização da Prova do Teorema de Existência de Unificadores Mais Gerais em Teorias de Primeira-Ordem

Por

Andréia Borges Avelar[1]

Orientador: Prof. Dr. Mauricio Ayala Rincón
Coorientador: Prof. Dr. André Luiz Galdino

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

A Hélio, Antéria e Elaine.

A Sabedoria é mais móvel que qualquer movimento
e, por sua pureza, tudo atravessa e penetra.
As virtudes são seus frutos;
ela ensina a temperança e a prudência,
a justiça e a fortaleza,
que são, na vida, os bens mais úteis aos homens.

Sb 7,24; 8,7b

Agradecimentos

Agradeço a Deus por ter me concedido forças para concluir este trabalho, consolo nos momentos difíceis, coragem e equilíbrio para vencer os obstáculos.

Agradeço à minha família, toda a compreensão e apoio, que sem dúvida foram fundamentais para a conclusão deste trabalho.

Agradeço de maneira especial ao meu orientador, Prof. Mauricio Ayala Rincón, por que foi paciente, me concedeu um voto de confiança, foi um excelente orientador e amigo.

Agradeço ao meu coorientador, Prof. André Luiz Galdino, a grande ajuda durante a realização deste trabalho e as valiosas dicas.

Agradeço os Profs. Flávio e Mário Benevides, as valiosas sugestões durante a correção deste trabalho.

Agradeço ao Prof. Célius, as conversas animadoras e por muitas vezes ter sido um amigo que soube ouvir.

Agradeço a Prof. Cátia, porque também acreditou que eu seria capaz de concluir este trabalho e me deu o seu apoio.

Agradeço ao amigo Martins e à amiga Luciene, que me concederam um apoio importante nos momentos mais difíceis.

Aos amigos João Marcelo e João Vítor, a companhia e incentivo durante os estudos para o exame de qualificação.

Agradeço ao amigo Vagner e à amiga Flávia, o apoio, as conversas descontraídas e a companhia tão agradável que tornou os estudos das matérias que fizemos juntos muito mais fáceis.

Agradeço ao amigo Wagner, porque sempre acreditou que eu concluiria este trabalho e foi um grande incentivador.

Agradeço à amiga Thaynara, a companhia tão alegre que foi muito motivadora nos momentos conclusivos deste trabalho.

Aos amigos Daniel, Daniele, Leonardo, Fábio, Ana Cristina e Kaliana, que tamém estiveram presentes nos últimos meses e com os quais passei momentos muito agradáveis.

Por fim, agradeço a todos os amigos, professores e funcionários do Departamento de Matemática da UnB, que de alguma forma contribuíram para a finalização deste trabalho.

Resumo

Neste trabalho apresenta-se uma formalização do teorema de existência de unificadores mais gerais em teorias de primeira ordem. Tal formalização foi desenvolvida na linguagem de especificação de ordem superior, do assistente de prova PVS. A prova mecânica é muito semelhante às provas encontradas em livros-texto, as quais se baseiam na correção do já conhecido algoritmo de unificação de Robinson de primeira ordem. A prova do teorema foi aplicada dentro de uma *teoria* completa, desenvolvida em PVS, para sistemas de reescrita de termos, a fim de obter uma formalização completa do Teorema dos Pares Críticos de Knuth-Bendix. Para chegar a esta formalização foi construída uma especificação em PVS de uma *teoria* para unificação de primeira ordem, onde foram formalizadas as propriedades de generalidade e terminação de uma versão do algoritmo de unificação de Robinson restrito a termos unificáveis.

Palavras-chave: Verificação formal, unificação de primeira ordem, unificador mais geral, PVS.

Abstract

This work presents the formalization of the theorem of existence of most general unifiers in first-order theories. The formalization was developed in the higher-order specification language, of the proof assistant PVS. The mechanical proof is very similar to that found in textbooks, which are based on proving the correction of the well-known Robinson's first-order unification algorithm. The proof of the theorem was applied within a complete *theory*, also developed in PVS, for term rewriting systems in order to obtain the full formalization of the Knuth-Bendix Critical Pair theorem. To reach this formalization, it was build in PVS a specification of a *theory* for first-order unification, where properties of generality and termination of a version of the Robinson's unification algorithm restricted to unifiable terms were formalized.

Keywords: Formal verification, first-order unification, most general unifier, PVS.

Lista de Tabelas

4.2.1 Construtor `resolving_diff` 45

4.2.2 Construtor `sub_of_frst_diff` 46

4.2.3 Construtor `unification_algorithm` 47

4.2.4 Lemas sobre `resolving_diff` 49

4.2.5 Lemas sobre `sub_of_frst_diff` 50

4.2.6 Principais lemas sobre `sub_of_frst_diff` 51

4.2.7 Lemas sobre `unification_algorithm` 52

5.0.1 Análise Quantitativa da *sub-teoria* `unification` 93

Lista de Figuras

2.2.1 Algoritmo de unificação de Robinson 14

2.2.2 Versão do algoritmo de unificação de Robinson 15

4.1.1 Estrutura hierárquica da *sub-teoria* `unification` 44

4.3.1 Árvore de prova do teorema `unification`. 54

4.4.1 Início da árvore de prova do lema `unification_algorithm_gives_unifier` 58

4.4.2 Ramo principal da árvore de prova do lema `unification_algorithm_gives_unifier` . 60

4.4.3 Início da árvore de prova do lema `unification_algorithm_gives_mg_subs` 65

4.4.4 Parte da árvore de prova do lema `unification_algorithm_gives_mg_subs` 67

4.4.5 Parte da árvore de prova do lema `unification_algorithm_gives_mg_subs` 72

4.5.1 Início da árvore de prova do lema `vars_ext_sub_of_frst_diff_decrease` 76

4.5.2 Parte da árvore de prova do lema `vars_ext_sub_of_frst_diff_decrease` 78

4.5.3 Parte da árvore de prova do lema `vars_ext_sub_of_frst_diff_decrease` 80

4.5.4 Início da árvore de prova do lema `sub_of_frst_diff_unifier_o` 82

4.5.5 Parte final da árvore de prova do lema `sub_of_frst_diff_unifier_o` 85

4.5.6 Parte final da árvore de prova do lema `sub_of_frst_diff_unifier_o` 88

Índice

Resumo vi

Abstract vii

Lista de Tabelas viii

Lista de Figuras ix

1 Introdução 1

1.1 Motivação 1

1.2 Organização 2

2 Unificação de Primeira Ordem 4

2.1 Unificação: Visão Informal do Problema, História e Aplicações 4

2.1.1 Visão Informal do Problema de Unificação 4

2.1.2 História e Aplicações 5

2.2 O Problema de Unificação 8

2.2.1 Assinaturas, Termos e Substituições 8

2.2.2 O Problema de Unificação e as Substituições mais Gerais 12

2.2.3 O Algoritmo de Unificação 14

3 Semântica do PVS **26**

3.1 A Linguagem de Especificação do PVS 26

3.2 O Assistente de Provas 27

3.3 A Checagem de Tipos em PVS 28

3.4 As Regras de Prova do PVS 32

3.4.1 Regras Estruturais 32

3.4.2 Regra de Corte 33

3.4.3 Regras para Axiomas Proposicionais 33

3.4.4 Regras de Contexto 34

3.4.5 Regras Condicionais 34

3.4.6 Regras de Igualdade 35

3.4.7 Regras de Igualdade Booleana 35

3.4.8 Regras de Redução 35

3.4.9 Regras de Extensionalidade 36

3.4.10 Regra de Restrição de Tipo 36

3.5 *Sub-teorias* da *Teoria* `TRS` 38

3.5.1 A *Sub-teoria* `term` 38

3.5.2 A *Sub-teoria* `positions` 39

3.5.3 A *Sub-teoria* `subterm` 40

3.5.4 A *Sub-teoria* `substitution` 41

4 Formalização da Teoria de Unificação **43**

4.1 Estrutura Hierárquica da Teoria `unification` 43

4.2 Organização da Teoria `unification` 44

4.3 Formalização do Teorema Sobre a Existência de mgu's 53

4.4 Formalização dos Lemas Sobre o Construtor `unification_algorithm` 57

4.4.1 Lema `unification_algorithm_gives_unifier` 58

4.4.2 Lema `unification_algorithm_gives_mg_subs` 65

4.5 Terminação e Formalização de Lemas Sobre o Construtor `sub_of_frst_diff` . . 74

4.5.1 Lema `vars_ext_sub_of_frst_diff_decrease` 75

4.5.2 Lema `sub_of_frst_diff_unifier_o` 81

4.5.3 Lema `ext_sub_of_frst_diff_unifiable` 89

5 Conclusão e Trabalhos Futuros **92**

5.1 Trabalhos Relacionados 93

5.2 Trabalhos Futuros 95

A O Código da Especificação **97**

B Formalização do lema `sub_of_frst_diff_remove_x` **103**

Referências Bibliográficas **113**

Capítulo 1

Introdução

Neste trabalho apresentamos uma especificação no assistente de prova PVS, de uma versão do algoritmo de unificação de Robinson, bem como a formalização da correção deste algoritmo. Como resultado da correção do algoritmo, formaliza-se um teorema que estabelece a existência de unificadores mais gerais para dois termos unificáveis, em sistemas de primeira ordem. Para este resultado, já bem estabelecido em computação, encontramos várias aplicações em lógica computacional, que vão deste a completude do princípio de resolução de primeira ordem [26] à correção do procedimento de completação de Knuth-Bendix [14] para sistemas de reescrita de termos, além de aplicações em linguagens de programação, como mecanismos de inferência de tipos. Esta especificação/formalização consiste de um desenvolvimento de uma *teoria* em PVS a qual denominamos `unification`.

1.1 Motivação

O desenvolvimento de uma *teoria* em PVS para tratar da existência de unificadores mais gerais foi motivado pela formalização de uma biblioteca em PVS para sistemas de reescrita de termos [8], a *teoria* `trs`, em que o resultado que estabelece a existência e unicidade de unificadores mais gerais era tratado como um axioma.

A *teoria* `trs` foi desenvolvida com o objetivo de fornecer uma formalização de conceitos básicos, através dos quais fosse possível especificar e formalizar outros conceitos e resultados da teoria de sistemas de reescrita de termos. Contudo a *teoria* `trs` estava incompleta, no sentido de que possuia um resultado não formalizado. Assim, o objetivo inicial deste

trabalho foi o de obter uma formalização de um teorema que garantisse a existência de unificadores mais gerais para termos de primeira ordem e com isto ter uma *teoria* em PVS para sistemas de reescrita de termos completa. Com este objetivo foi realizado o desenvolvimento da *sub-teoria* `unification`.

A *teoria* `trs` é composta por um conjunto de *sub-teorias*. Assim, para chegar a uma formalização do teorema que estabelece a existência de unificadores mais gerais a *teoria* `unification` foi desenvolvida como uma *sub-teoria* da *teoria* `trs`. Isto significa que a *sub-teoria* `unification` importa algumas *sub-teorias* da *teoria* `trs`, onde encontramos especificações de conceitos básicos, como por exemplo as definições de termos, posições, substituições, etc., e a formalização de vários resultados. Assim, nas *teorias* importadas encontramos uma base teórica completa para o desenvolvimento de uma *teoria* sobre unificação.

1.2 Organização

No decorrer desta apresentação quando usamos a palavra *teoria* ou *sub-teoria* em itálico, estamos nos referindo à *especificação* de uma teoria desenvolvida no assistente de prova PVS, e quando usamos *formalização* estamos nos referindo à prova mecânica feita em PVS de algum resultado ou teorema.

No Capítulo 2 apresentamos a teoria de unificação de primeira ordem, que envolve a introdução e definição de conceitos tratados em unificação como termos, substituições, unificadores mais gerais, etc., além de apresentarmos uma versão do algoritmo de unificação de Robinson, para em seguida verificar analiticamente a correção e completude deste algoritmo. No Capítulo 3 apresentamos uma visão geral da semântica do assistente de prova PVS, e expomos as especificações, feitas anteriormente em PVS no trabalho de Galdino e Ayala-Rincón [8], dos conceitos e definições apresentados no Capítulo 2. No Capítulo 4 apresentamos a organização da *teoria* `unification`, os principais aspectos da especificação e a formalização dos teoremas da *teoria*, que são aqueles onde verificamos a correção do algoritmo de unificação e a existência de unificadores mais gerais para termos unificáveis, além destes apresentamos a formalização de alguns lemas importantes para a

verificação da correção do algoritmo. Em seguida, apresentamos a conclusão e trabalhos futuros, além de alguns trabalhos relacionados. No Apêndice A, apresentamos o código da especificação da *sub-teoria* `unification`, mas o seu desenvolvimento completo encontra-se disponível em `http://ayala.mat.unb.br/publications.html`, juntamente com a *teoria* `trs`. No Apêndice B, apresentamos um exemplo de uma formalização detalhada de um dos lemas da *sub-teoria* `unification`.

Acreditamos que esta seja a primeira formalização completa de uma teoria em PVS para unificação de primeira ordem.

Capítulo 2

Unificação de Primeira Ordem

O problema de unificação tem sido estudado em várias áreas da ciência da computação, incluindo dedução automática, programação lógica, complexidade computacional, entre outras. Neste capítulo apresentamos o conceito de unificação de uma maneira informal, seguimos expondo um levantamento histórico e algumas aplicações mostrando onde o problema de unificação foi originalmente introduzido, e então finalizamos apresentando o problema de unificação de maneira formal.

2.1 Unificação: Visão Informal do Problema, História e Aplicações

2.1.1 Visão Informal do Problema de Unificação

Unificação é um processo fundamental sobre o qual vários métodos de dedução automática são baseados. A teoria que envolve o problema de unificação surge da necessidade de formalizar aplicações específicas deste processo. Esta teoria provê definições para noções importantes como *instanciação*, *termos unificáveis*, *unificadores mais gerais*, etc., investiga propriedades destas noções, além de buscar e analizar algoritmos de unificação que podem ser utilizados em vários contextos.

Em muitas aplicações de unificação o interesse não está apenas em responder o problema de decisão para unificação, isto é, dizer "sim" ou "não" para responder se dois termos s e t são unificáveis. Se estes dois termos são unificavéis, busca-se também uma solução, isto é, uma substituição que torne estes dois termos idênticos. Tal substituição é chamada

um unificador de s e t. Em geral um problema de unificação pode ter várias soluções, mas felizmente nas várias aplicações de unificação, o interesse não está em encontrar todos os unificadores para um determinado problema, mas sim em determinar um *unificador mais geral*, isto é, um unificador a partir do qual obtem-se todos os outros por *instanciação*.

Exemplo 2.1.1: Considere os termos $f(x, y)$ e $f(y, x)$. Note que estes dois termos podem ser unificados substituindo-se x e y pelo mesmo termo s, e como existe uma infinidade de termos possíveis, temos que este problema possui infinitas soluções. Contudo a substituição dada por $\theta := \{x/y\}$ é um *unificador mais geral* do problema, visto que para qualquer termo s, temos que $\{x/s, y/s\} = \{y/s\} \circ \theta$.

Portanto, um algoritmo de unificação não deve responder apenas se um dado problema de unificação tem ou não solução, além disso, se espera que o algoritmo compute um *unificador mais geral* para o problema.

Até este ponto falamos apenas de unificação *sintática* de termos de *primeira ordem*, isto significa que os termos devem ser sintaticamente iguais e que não temos variáveis de segunda ordem, isto é, variáveis para funções. Por exemplo, os termos $f(x, y)$ e $h(x, y)$ obviamente não podem ser sintaticamente iguais em unificação de primeira ordem, contudo se H é uma variável de segunda ordem, os termos $f(x, y)$ e $H(x, y)$ podem ser sintaticamente iguais, via substituições, em unificação de segunda ordem. Mas neste trabalho estamos interessados apenas em unificação de primeira ordem.

2.1.2 História e Aplicações

Baseamos esta seção nos *surveys* de Knight [13] e de Baader e Snyder [3], que utilizamos como fonte para as referências bibliográficas que citamos.

Em 1930, Jacques Herbrand [10] apresentou em sua tese de doutorado um algoritmo não determinístico para computar um unificador de dois termos. Mas foi em 1965, que o nome unificação e a primeira investigação formal sobre o assunto aparecem no trabalho de Robinson [26], que introduziu unificação como sendo a operação básica para o seu princípio de resolução, mostrando que termos unificáveis possuem um unificador mais geral; além de descrever um algoritmo, veja Figura 2.2.1, para computar tal unificador e provar que

este algoritmo de fato computa um unificador mais geral para um conjunto unificável de expressões bem formadas. Em 1964, Jim Guard [9] estudava independentemente o problema de unificação sob o nome de *matching* e cinco anos depois, em 1970, Reynolds [25] discutiu termos de primeira ordem usando teoria de reticulado e mostrou que também existe uma única *generalização mais específica* de quaisquer dois termos unificáveis. Em 1970, segundo Baader e Snyder [3], as noções de *unificação* e *unificador mais geral* foram independentemente reinventadas por Knuth e Bendix [14] como uma ferramenta para testar confluência local de sistemas de reescrita de termos através de pares críticos.

A versão original do algoritmo de unificação de Robinson é ineficiente, pois é exponencial em tempo de execução e em espaço. Por isso surgiu um grande interesse em obter algoritmos de unificação eficientes. O próprio Robinson passou a pesquisar sobre a eficiência na unificação, e argumentou em [27] que uma representação mais concisa para os termos era necessária. Com sua nova formulação, Robinson conseguiu uma grande melhoria na complexidade de espaço exigida por seu algoritmo de unificação.

Assim, em pesquisas sobre complexidade computacional, surgem vários trabalhos em busca de eficiência no processo de unificação. Em 1972, Boyer e Moore [4] apresentam um algoritmo de unificação que divide a estrutura dos termos, este algoritmo era eficiente em espaço, mas ainda exponencial em tempo de execução. Em 1975, Venturini-Zilli [31] consegue reduzir a complexidade de tempo do algoritmo de unificação de Robinson para tempo quadrático. Em 1976, no seu trabalho sobre unificação de ordem superior [11], Huet apresenta um procedimento baseado em classes de equivalência de subtermos, que era quase linear em tempo de execução. Ainda em 1976, Paterson e Wegman [23], descobrem um algoritmo de unificação realmente linear, baseado sobre um método que consistia na propagação da relação de classes de equivalência do algoritmo de Huet. Em 1976, Martelli e Montanari [16], independentemente descobrem outro algoritmo de unificação linear. Eles chegam a este algoritmo a partir da estrutura de termos proposta por Boyer e Moore. Mas só em [17], no ano de 1982, é que Martelli e Montanari apresentam uma descrição completa de um algoritmo de unificação eficiente, mas este último algoritmo não era realmente linear. Em 1983, Corbin e Bidoit [5] reabilitam o algoritmo de unificação de Robinson usando novas estruturas de dados. Eles conseguem reduzir a complexidade

de tempo do algoritmo de Robinson, que era exponencial, para $O(n^2)$, e afirmam que o algoritmo é mais simples que o proposto por Martelli e Montanari além de ser superior na prática.

O algoritmo de unificação de Robinson usa a representação de termos em lógica de primeira ordem dados por uma sequência de símbolos, que é uma representação bastante simples, onde os termos podem ser vistos como um *arranjo* linear. Esta representação por sequência de símbolos é equivalente à representação por árvores. Mas este tipo de representação é mais útil quando os termos não assumem formas muito complicadas. De fato, nesta representação pode ser necessário gerar estruturas de termos exponencialmente grandes durante o processo de unificação.

No sentido de contornar este problema, alguns algoritmos de unificação usam uma representação de termos por grafos, chamamos esta representação de grafo acíclico direto, ou pela sigla em inglês DAG, como é mais comumente chamada. Esta abordagem dos termos contorna o problema de duplicação de subtermos gerado por substituições, através de uma representação por grafos que podem dividir estruturas. Em tal representação, todos os vértices do grafo são rotulados. A idéia central é não permitir que subtermos idênticos apareçam em posições distintas de um termo. Isto é feito através de uma atualização de ponteiros, que passam a indicar um outro subtermo depois de uma instanciação. Podemos ter mais de um ponteiro direcionado para uma mesma estrututra, isto consiste em dividir estruturas. Um algoritmo que utiliza tal estrutura de dados, necessita de uma estrutura adicional para ponteiros que ligam a cada variável um termo que seja sua instância por uma substituição. Com esta representação para termos, a complexidade do algoritmo passa a ser quadrática. De fato, Corbin e Bidoit utilizam esta representação para os termos, a fim de reduzir a complexidade de tempo do algoritmo de unificação de Robinson para tempo quadrático.

2.2 O Problema de Unificação

Nesta seção apresentamos o problema de unificação analiticamente, isto é, expomos uma análise algébrica do problema, primeiro introduzindo as definições e a notação que utilizamos para tratar do problema de unificação, em seguida definimos o problema de unificação em si, para então apresentar a versão do algoritmo de unificação de Robinson que utilizamos na formalização proposta por este trabalho. Concluímos provando analiticamente a correção deste algoritmo. Na apresentação desta seção, utilizamos as notações e definições de [2] e de [1].

2.2.1 Assinaturas, Termos e Substituições

Nas seções anteriores já temos introduzido alguns elementos presentes na teoria que envolve o problema de unificação, sem definí-los formalmente. Alguns destes elementos, são o que chamamos de *termos*, *substituições*, *unificadores*, por exemplo. Passemos agora a uma definição formal destes elementos. Primeiro introduzimos a seguinte notação para variáveis, constantes e símbolos de função:

- Variáveis: $\{u, v, w, x, y, z\}$,
- Constantes: $\{a, b, c\}$,
- Símbolos de função: $\{f, g, h\}$.

Os termos são construídos a partir de símbolos de função, associados a um natural n que representa a sua aridade, e variáveis. Por exemplo, se g é um símbolo de função de aridade 3 e u, v e w são variáveis, então $g(u, v, w)$ é um termo. Então, antes de definirmos formalmente o que é um termo, vamos introduzir a noção de assinatura.

Definição 2.2.1: Uma **assinatura** Σ é um conjunto de **símbolos de função**, onde a cada $f \in \Sigma$ é associado a um número natural n, que chamamos a **aridade** de f. Para cada $n \geqslant 0$, denotamos o conjunto de todos os elementos n-ários de Σ por Σ^n. Os elementos de Σ^0 são **símbolos de constante**, isto é, as constantes são funções com aridade 0.

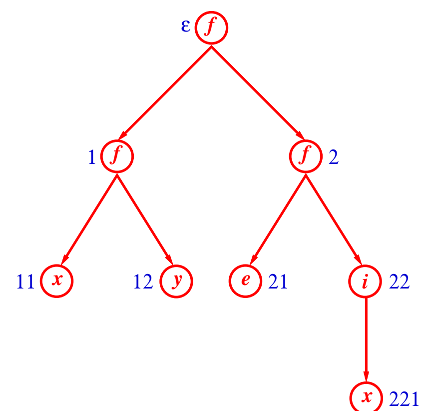
Por exemplo, se queremos considerar um *grupo* G, que é um conjunto não vazio de elementos, munido de uma operação binária associativa, que chamamos de produto; de uma operação unária para denotar o inverso de um elemento de G; além do elemento neutro, usamos a seguinte assinatura: $\Sigma_G := \{e, i, f\}$, onde e é uma constante, isto é, uma função de aridade 0, que denota o elemento neutro, i é um símbolo de função unário e f é um símbolo de função binário.

Tendo em mãos a definição de assinatura, podemos agora definir o que vem a ser um termo. Lembrando que estamos tratando sempre de termos de primeira ordem. A noção de termo é definida recursivamente da seguinte forma:

Definição 2.2.2: Seja Σ uma assinatura e $\mathcal{V}$ um conjunto de variáveis, então o conjunto $T(\Sigma, \mathcal{V})$ de todos os **Σ-termos**, ou simplesmente termos, sobre $\mathcal{V}$ é dado por:

- $\mathcal{V} \subset T(\Sigma, \mathcal{V})$, isto é, toda variável é um termo;
- Se $f \in \Sigma^n$ é um símbolo de função de aridade $n \geqslant 0$ e $t_1, \ldots, t_n \in T(\Sigma, \mathcal{V})$ são termos, então $f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{V})$ é um termo, isto é, para $n = 0$ temos que toda constante é um termo, e toda aplicação de símbolos de função de aridade $n > 0$ em termos, é ainda um termo.

Por exemplo, considerando a assinatura $\Sigma_G = \{e, i, f\}$, temos que se $x, y \in G$ são elementos de G, então $f(f(x, y), f(e, i(x)))$ é um Σ_G-termo, isto é, $f(f(x, y), f(e, i(x))) \in T(\Sigma_G, G)$. Encontramos na representação por árvores uma forma bastante prática de representar um termo. Para ilustrar veja ao lado a representação por árvore do termo $f(f(x, y), f(e, i(x)))$.



Note que na figura acima utilizamos a numeração por sequências de inteiros positivos para os nós da árvore. Com esta numeração podemos nos referir as posições do termo. Veja por exemplo que o nó raiz está rotulado com ε, que na nossa notação representa a sequência vazia, e se refere ao primeiro símbolo de função f que aparece no termo. O

primeiro nó a esquerda se refere ao subtermo na posição 1 do termo principal, que é dado por $f(x, y)$. A seguir, temos algumas definições acerca de termos.

Nas definições seguintes sejam Σ uma assinatura, $\mathcal{V}$ um conjunto de variáveis e s e t termos de $T(\Sigma, \mathcal{V})$.

Definição 2.2.3: O conjunto de **posições** do termo s, denotado por $\mathcal{P}os(s)$, é um conjunto de sequências de números naturais, que é definido indutivamente como segue:

- Se $s \in \mathcal{V}$, então $\mathcal{P}os(s) = \varepsilon$.
- Se $s = f(s_1, \ldots, s_n)$, então

$$\mathcal{P}os(s) := \{\varepsilon\} \ \cup \ \bigcup_{i=1}^{n} \{ip \mid p \in \mathcal{P}os(s_i)\}.$$

O comprimento de um termo s, denotado por $|s|$, é a cardinalidade do conjunto $\mathcal{P}os(s)$.

Definição 2.2.4: Para cada $p \in \mathcal{P}os(s)$, o **subtermo** de s na posição p, denotado por $s|_p$, é definido por indução no comprimento de p como segue:

- $s|_\varepsilon := s$, para $p = \varepsilon$.
- $f(s_1, \ldots, s_n)|_{iq} := s_i|_q$, para $p = iq$.

Note que, para $p = iq$, se $p \in \mathcal{P}os(s)$, então s é da forma $f(s_1, \ldots, s_n)$ com $i \leqslant n$.

Definição 2.2.5: O conjunto de **variáveis que ocorrem em um termo** s, denotado por $\mathcal{V}ars(s)$, é dado por:

$$\mathcal{V}ars(s) := \{x \in \mathcal{V} \mid \exists p \in \mathcal{P}os(s) \text{ tal que } s|_p = x\}.$$

Dizemos que $p \in \mathcal{P}os(s)$ é uma posição de variável do termo s, se $s|_p$ é uma variável.

Para falar de unificação ainda precisamos introduzir a noção de substituição. A título de notação, utilizaremos letras gregas minúsculas $\{\sigma, \alpha, \beta, \ldots\}$, para denotar substituições.

Definição 2.2.6: Seja Σ uma assinatura e $\mathcal{V}$ um conjunto enumerável de variáveis. Uma **$\mathbf{T}(\boldsymbol{\Sigma}, \mathcal{V})$-substituição**, ou simplesmente substituição, é uma função $\sigma : \mathcal{V} \rightarrow T(\Sigma, \mathcal{V})$ tal que:

(a) $\sigma(x) \neq x$, somente para um número finito de variáveis.

(b) O conjunto de variáveis para as quais $\sigma(x) \neq x$, é chamado o **domínio** de σ, e denotado por:

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

(c) A **imagem** de σ é o conjunto composto por todo termo s tal que $s = \sigma(x)$ para algum x no domínio de σ. Denotamos tal conjunto por:

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

A substituição σ para a qual $Dom(\sigma) = \emptyset$, isto é, $\sigma(x) = x$, $\forall x \in \mathcal{V}$, denominamos substituição *identidade* e a denotaremos por id.

Note que, pela Definição 2.2.6, temos que uma substituição tem domínio finito. Assim, podemos usar a seguinte notação de conjunto

$$\sigma := \{x_1/r_1, \ldots, x_n/r_n\},$$

para denotar uma substituição σ cujo domínio seja dado por $Dom(\sigma) = \{x_1, \ldots, x_n\}$ e cuja imagem seja dada por $Ran(\sigma) = \{r_1, \ldots, r_n\}$.

Observamos que uma substituição σ pode ser extendida homeomorficamente ao conjunto de termos $T(\Sigma, \mathcal{V})$, por uma aplicação $\hat{\sigma} : T(\Sigma, \mathcal{V}) \rightarrow T(\Sigma, \mathcal{V})$, da seguinte forma:

- Para $s = x \in \mathcal{V}$, definimos $\hat{\sigma}(s) := \sigma(x)$.
- Para $s = f(s_1, \ldots, s_n)$, definimos $\hat{\sigma}(s) := f(\hat{\sigma}(s_1), \ldots, \hat{\sigma}(s_n))$.

No processo proposto pelo algoritmo de unificação são feitas composições entre substituições até que se chegue a uma solução do problema. Assim, é preciso definir o que vem a ser composição de substituições.

Definição 2.2.7: Sejam $\sigma := \{x_1/r_1, \ldots, x_n/r_n\}$ e $\alpha := \{y_1/s_1, \ldots, y_m/s_m\}$ substituições. Então definimos a **composição** $\sigma \circ \alpha$ como sendo a substituição:

$$\sigma \circ \alpha \ := \ \{y_1/\hat{\sigma}(s_1), \ldots, y_n/\hat{\sigma}(s_n), x_1/r_1, \ldots, x_n/r_n\},$$

eliminando qualquer ligação $y_j/\hat{\sigma}(s_j)$, para a qual $y_j = \hat{\sigma}(s_j)$ e qualquer ligação x_i/r_i, para a qual $x_i \in \{y_1, \ldots, y_m\}$.

2.2.2 O Problema de Unificação e as Substituições mais Gerais

Até este ponto todas as definições e notações colocadas, tiveram o objetivo de preparar a base para podermos falar do que é o nosso principal interesse, o problema de unificação. Já colocamos na primeira seção deste capítulo que o problema de unificação é mais geralmente tratado no seguinte contexto: dados dois termos quaisquer queremos saber se existe uma substituição que torna os dois termos idênticos e estudar formas de obter tal substituição bem como as *propriedades matemáticas* da mesma. De maneira mais formal, apresentamos na Definição 2.2.8 em que consiste o processo de unificação envolvendo dois termos quaisquer, mais a frente na Definição 2.2.10, falamos do problema de unificação.

Note que a definição a seguir é construída sobre a noção de *teoria equacional*, que nada mais é do que um conjunto de pares (s, t), onde s e t são termos de $T(\Sigma, \mathcal{V})$, tais que s e t pertencem a uma mesma classe de equivalência induzida por um conjunto de Σ-identidades E em $T(\Sigma, \mathcal{V})$. Em notação matemática:

$$\approx_E := \{(s,t) \in T(\Sigma, \mathcal{V}) \times T(\Sigma, \mathcal{V}) \mid E \vDash s \approx t\}.$$

Definição 2.2.8: Unificação é o processo de resolver o seguinte problema de satisfazibilidade: Dada uma teoria equacional E e dois termos s e t em $T(\Sigma, \mathcal{V})$, encontrar uma substituição σ tal que $\hat{\sigma}(s) \approx_E \hat{\sigma}(t)$. Dizemos que σ é um **unificador** de s e t, e $\sigma \in \mathcal{U}(s, t)$, onde $\mathcal{U}(\mathbf{s}, \mathbf{t})$ denota o conjunto formado por todos os unificadores dos termos s e t.

Por exemplo, considere os termos $s = f(a, x)$ e $t = f(y, h(b))$. Tais termos são ditos unificáveis, pois substituir x por $h(b)$ e y por a torna-os idênticos e iguais a $f(a, h(b))$. Assim, neste caso um unificador de s e t é dado por $\sigma := \{x/h(b), y/a\}$.

Neste trabalho, nos concentramos no caso em que a teoria equacional E é vazia. Este caso é do nosso interesse pois é neste contexto que se fala sobre o teorema dos pares críticos de Knuth-Bendix, formalizado na teoria `trs`, onde a existência de unificadores mais gerais foi originalmente axiomatizada, e que visamos completar. Quando $E = \emptyset$ o processo de unificação recebe o nome especial de *unificação sintática*.

Para estudarmos as propriedades das substituições computadas por um algoritmo de

unificação, precisamos ainda de algumas noções importantes e que nos interessam acerca de substituições. Como já colocamos anteriormente, em geral não se está interessado em um algoritmo que compute todos os unificadores de um dado problema, mas sim em um unificador especial, um que seja mais geral. Assim, definimos o que vem a ser uma substituição mais geral.

Definição 2.2.9: Uma substituição σ é **mais geral** que uma substituição σ' se existe uma substituição δ tal que $\sigma' = \delta \circ \sigma$. Neste caso dizemos que σ' é uma **instância** de σ e denotamos este fato por $\sigma \lesssim \sigma'$, onde $\lesssim$ é uma relação sobre substituições que é uma pré-ordem.

Agora podemos definir o que é um problema de unificação.

Definição 2.2.10: Uma instância do **problema de unificação** é um conjunto finito de equações $S = \{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$. O problema é determinar se existe uma substituição σ tal que $\sigma(s_i) = \sigma(t_i)$, para todo $i = 1, \ldots, n$.

Note que na definição de problema de unificação, temos n equações, envolvendo pares de termos que devemos responder se são unificáveis e em seguida buscar uma substituição que seja um unificador de todos os n pares de termos. Mas o algoritmo de unificação apresentado na Seção 2.2.3, tem como parâmetros apenas um par de termos, isto é, o algoritmo apresentado busca responder, para dois termos s e t, se existe uma substituição σ tal que $\hat{\sigma}(s) = \hat{\sigma}(t)$. Contudo, o algoritmo pode ser extendido a uma instância do problema de unificação como definido acima.

Definição 2.2.11: Dizemos que uma substituição σ é um **unificador** ou **solução** de um problema de unificação S, se $\hat{\sigma}(s_i) = \hat{\sigma}(t_i)$ para todo $s_i \stackrel{?}{=} t_i \in S$, $i = 1, \ldots, n$. Denotamos ainda por $\mathcal{U}(S)$ o conjunto de todos os unificadores de S, e dizemos que S é unificável se $\mathcal{U}(S) \neq \emptyset$.

Agora podemos definir o que é um unificador mais geral de um problema de unificação. Por simplicidade, de agora em diante vamos fazer uso da sigla *mgu*, que vem da expressão em inglês *most general unifier*, para significar que estamos falando de um unificador mais geral.

Início	*Unification Algorithm(A: Conjunto não vazio de expressões bem formadas)*
Passo 1.	*Defina $\sigma_0 = id$ e $k = 0$, e vá para o passo 2.*
Passo 2.	*Se $A\sigma_k$ não é unitário, vá para o passo 3. Caso contrário, defina $\sigma_A = \sigma_k$ e termine.*
Passo 3.	*Defina V_k como sendo o menor, e U_k como o sendo o próximo, na orden lexicográfica do conjunto de diferenças B_k de $A\sigma_k$. Se V_k é uma variável, e não ocorre em U_k, defina $\sigma_{k+1} = [V_k/U_k] \circ \sigma_k$, adicione 1 a k, e retorne ao passo 2. Caso contrário, termine.*
Fim	

Figura 2.2.1: Algoritmo de unificação de Robinson original encontrado em [26].

Definição 2.2.12: Dizemos que uma substituição σ é um **mgu** de um problema de unificação S se:

- $\sigma \in \mathcal{U}(S)$ e
- Para todo $\sigma' \in \mathcal{U}(S)$ vale que $\sigma \lesssim \sigma'$.

Em outras palavras, uma substituição σ é um mgu de um problema de unificação S, se é um unificador de S e se para qualquer outra substituição σ' que seja também unificador de S, existe uma substituição δ tal que $\sigma' = \delta\sigma$.

2.2.3 O Algoritmo de Unificação

Agora temos material suficiente para apresentar a versão do algoritmo de unificação de Robinson, em seguida provar a correção e a completude deste algoritmo. Na Figura 2.2.2 temos este algoritmo. Apresentamos também na Figura 2.2.1 o algoritmo original de Robinson, extraído de [26], para que se possa fazer uma comparação entre os dois e ver que consistem do mesmo processo.

Antes de partirmos para a verificação da correção da versão do algoritmo de unificação de Robinson, o algoritmo *Robinson-Unification*, proposto na Figura 2.2.2, vamos enfatizar

```
01:  k := 0 (variável global)
     σ_0 := id
02:  BEGIN Robinson-Unification(s, t)
03:     IF s = t THEN σ_{k+1} := id
04:     ELSE
05:        p ← posição da "primeira diferença" entre os termos s e t
                 (tomada mais externamente e mais à esquerda)
06:        IF s|_p ∉ V e t|_p ∉ V THEN fail
07:        ELSE
08:           IF s|_p ∈ V e s|_p ∈ V(t|_p) THEN fail
09:           ELSE
10:              IF t|_p ∈ V e t|_p ∈ V(s|_p) THEN fail
11:              ELSE
12:                 IF s|_p ∈ V THEN x := s|_p e r := t|_p
13:                 ELSE x := t|_p e r := s|_p
14:     σ_{k+1} := {x/r} (substituição que resolve a primeira diferença)
15:     k := k + 1
16:     return Robinson-Unification(σ̂_k(s), σ̂_k(t)) ∘ σ_k
17:  END
```

Figura 2.2.2: Versão do Algoritmo de Unificação de Robinson, onde k é iniciado como zero e σ_0 como a identidade

a notação que será adotada de agora em diante, para termos uma referência desta notação e evitar confusões. Como sugere o algoritmo de unificação, adotamos a seguinte notação:

- σ_k, para $k \in \{0, \ldots, n, \ldots\}$, denota um conjunto de variáveis globais, que indicam a ligação feita no k-ésimo passo do algorimo de unificação, sendo que σ_0 é iniciada globalmente como a substituição identidade;
- k é uma variável global de controle das chamadas recursivas do algoritmo, iniciada globalmente como zero;
- σ'_k indica a substituição iterada, computada até o k-ésimo passo do algoritmo de unificação, usada na demonstração;

- σ indica a substituição final, obtida pelo algoritmo de unificação, para dois termos unificáveis s e t, usada nas provas.

Observe que, como já mencionamos anteriormente, no algoritmo da Figura 2.2.2, nos concentramos no caso em que o conjunto de equações S, que define problema de unificação, é unitário. Isto é, $S := \{s \stackrel{?}{=} t\}$. Note também que a cada iteração, o algoritmo de unificação tem um novo conjunto de termos de entrada, obtido a partir do anterior e da substituição gerada pelo passo anterior. A fim de que o algoritmo esteja correto, desejamos que estes novos termos também sejam unificáveis, caso s e t o sejam. Observe ainda que o algoritmo é um processo que sempre termina, como provaremos adiante, pois a cada seleção da variável x e do termo r, substitui-se todas as ocorrências de x nos termos s e t pelo termo r, e como r não possui ocorrências de x, temos que a cardinalidade do conjunto dado pela união dos conjuntos de variáveis dos termos em questão, diminui a cada iteração do algoritmo, e como a união dos conjuntos de variáveis dos termos s e t é um conjunto finito, temos que o algoritmo termina. Em seguida vamos enunciar e demonstrar cada um destes fatos. Começamos pelo seguinte lema auxiliar:

Lema 2.2.13: *Uma equação $x \stackrel{?}{=} r$, onde $x \in \mathcal{V}ars(r)$ e $x \neq r$, não tem solução.*

Demonstração: Observe que não faz sentido considerar o caso em que o termo r é uma variável, pois caso r seja uma variável deve ser, por hipótese, diferente de x. Contudo temos também que x pertence a $\mathcal{V}ars(r)$. Logo, se r é uma variável deve ser igual a x, o que é uma contradição. Portanto, suponha que $r = f(r_1, ..., r_n)$ e que existe uma substituição σ tal que $\hat{\sigma}(x) = \hat{\sigma}(r)$. Assim,

$$\begin{aligned} x \in \mathcal{V}ars(r) &\Rightarrow \exists p \in \mathcal{P}os(r) \text{ tal que } r|_p = x. \\ &\Rightarrow \hat{\sigma}(r|_p) = \hat{\sigma}(x) = \hat{\sigma}(r). \\ &\Rightarrow \hat{\sigma}(r|_p) = \hat{\sigma}(r) \\ &\Rightarrow (\hat{\sigma}(r))|_p = \hat{\sigma}(r) \\ &\Rightarrow p = \epsilon. \end{aligned}$$

Mas se $p = \epsilon$, então $r|_p = r$. Contudo, $r|_p = x$. Logo, $r = x$. O que é um absurdo, pois por hipótese $r \neq x$. Portanto, temos que não existe uma substituição σ tal que $\hat{\sigma}(x) = \hat{\sigma}(r)$, isto é, nas condições das hipóteses do lema, a equação $x \stackrel{?}{=} r$, não tem solução. ∎

Nos dois lemas seguintes vamos demonstrar fatos fundamentais para a demonstração do Teorema 2.2.18, onde provamos que a substituição computada pelo algoritmo de unificação, tomando como entradas dois termos unificáveis s e t, é de fato um *mgu* de s e t. No primeiro lema provamos que para dois termos unificáveis e diferentes, se tomamos a substituição que resolve a primeira diferença entre estes termos, onde a primeira diferença é tomada na posição mais externa e mais à esquerda dos termos, então as instâncias dos termos por esta substituição são ainda dois termos unificáveis. No segundo lema, provamos um resultado importante para garantir a terminação do algoritmo de unificação, provamos que a união dos conjuntos de variáveis dos termos obtidos por instanciação nas chamadas recursivas do algoritmo de unificação sempre diminui. Precisamente, a cada iteração do algoritmo este conjunto passa a ter uma variável a menos: aquela capturada na iteração imediatamente anterior do algoritmo de unificação.

Lema 2.2.14: (Preservação da Generalidade) *Sejam s e t dois termos unificáveis e σ_k, para $k \neq 0$, a ligação computada pelo algoritmo de unificação no k-ésimo passo. Note que σ_k é a substituição que resolve a primeira diferença entre os termos $\hat{\sigma}'_{k-1}(s)$ e $\hat{\sigma}'_{k-1}(t)$, onde σ'_{k-1} é a substituição iterada computada pelo algoritmo de unificação até o $(k-1)$-ésimo passo. Então, $\hat{\sigma}'_{k-1}(s)$ e $\hat{\sigma}'_{k-1}(t)$ são unificáveis e*

$$\forall\theta \in \mathcal{U}(\ \hat{\sigma}'_{k-1}(s), \hat{\sigma}'_{k-1}(t)\ ),\ \ \exists\delta \text{ tal que } \theta = \delta \circ \sigma_k.$$

Demonstração: Vamos mostrar que a assertiva do lema é de fato verdadeira para $\delta = \theta$, isto é, $\forall\theta \in \mathcal{U}(\hat{\sigma}'_{k-1}(s), \hat{\sigma}'_{k-1}(t)),\ \theta = \theta \circ \sigma_k$. No caso em $s = t$, temos que $\sigma_1 = id$ e é trivial que $\theta = \theta \circ id$. O caso em que $s \neq t$, segue por indução em k:

B.I.: Para $k = 1$: por hipótese temos que $\sigma_0 = id$ e os termos s e t são unificáveis. Além disso, $\forall\theta \in \mathcal{U}(s,t),\ \theta = \theta \circ \sigma_1$. De fato, se p é a posição onde ocorre a primeira diferença entre os termos s e t, e se $s|_p$ é uma variável que não ocorre em $t|_p$, então a seguinte afirmação é verdadeira: $\forall\theta \in \mathcal{U}(s,t)$ e $\forall y \in \mathcal{V},\ \theta(y) = (\theta \circ \sigma_1)(y)$. Vamos analisar as possibilidades para a variável y:

$y = s|_p$**:** Neste caso, temos que:

$$\begin{aligned}
\theta(y) &= \theta(s|_p), \text{ pois } y = s|_p \\
&= \hat{\theta}(s|_p), \text{ pela observação da definição de substituição} \\
&= \hat{\theta}(t|_p), \text{ pois } \theta \in \mathcal{U}(s,t) \\
&= \hat{\theta}(\sigma_1(s|_p)), \text{ pois } \sigma_1 = \{s|_p/t|_p\} \\
&= (\theta \circ \sigma_1)(s|_p), \text{ por definição} \\
&= (\theta \circ \sigma_1)(y).
\end{aligned}$$

$y \neq s|_p$**:** Neste caso, temos diretamente que $\theta(y) = (\theta \circ \sigma_1)(y)$. Pois, como $y \neq s|_p$ temos que $y \notin Dom(\sigma_1)$, o que implica que $\sigma_1(y) = y$.

O caso em que $t|_p$ é uma variável que não ocorre em $s|_p$ é análogo.

P.I.: Suponha que a hipótese seja válida para k, vamos mostrar que também é válida para $k+1$, isto é, vamos mostrar que:

$$\hat{\sigma}'_{k-1}(s) \text{ e } \hat{\sigma}'_{k-1}(t) \text{ são unificáveis e}$$

$$\forall \theta \ \in \ \mathcal{U}(\ \hat{\sigma}'_{k-1}(s), \hat{\sigma}'_{k-1}(t)\ ),\ \theta = \theta \circ \sigma_k$$

$$\Downarrow$$

$$\hat{\sigma}'_k(s) \text{ e } \hat{\sigma}'_k(t) \text{ são unificáveis e}$$

$$\forall \theta \ \in \ \mathcal{U}(\ \hat{\sigma}'_k(s), \hat{\sigma}'_k(t)\ ),\ \theta = \theta \circ \sigma_{k+1}.$$

Primeiro, vamos verificar que os termos $\hat{\sigma}'_k(s)$ e $\hat{\sigma}'_k(t)$ são unificáveis. Isto equivale a mostrar que o conjunto $\mathcal{U}(\hat{\sigma}'_k(s), \hat{\sigma}'_k(t))$ é não vazio. Observe que $\sigma'_k = \sigma_k \circ \sigma'_{k-1}$, pois a substituição iterada obtida no k-ésimo passo do algoritmo de unificação é igual à composição da ligação obtida neste passo com a substituição iterada obtida no passo anterior. Assim,

$$\begin{aligned}
\theta \in \mathcal{U}(\hat{\sigma}'_{k-1}(s), \hat{\sigma}'_{k-1}(t)) &\Rightarrow \hat{\theta}(\hat{\sigma}'_{k-1}(s)) = \hat{\theta}(\hat{\sigma}'_{k-1}(t)) \\
&\Rightarrow (\widehat{\theta \circ \sigma_k})(\hat{\sigma}'_{k-1}(s)) = (\widehat{\theta \circ \sigma_k})(\hat{\sigma}'_{k-1}(t)) \\
&\Rightarrow \hat{\theta}((\widehat{\sigma_k \circ \sigma'_{k-1}})(s)) = \hat{\theta}((\widehat{\sigma_k \circ \sigma'_{k-1}})(t)) \\
&\Rightarrow \hat{\theta}(\hat{\sigma}'_k(s)) = \hat{\theta}(\hat{\sigma}'_k(t)) \\
&\Rightarrow \theta \in \mathcal{U}(\hat{\sigma}'_k(s), \hat{\sigma}'_k(t)).
\end{aligned}$$

Portanto, sabendo que o conjunto $\mathcal{U}(\hat{\sigma}'_k(s), \hat{\sigma}'_k(t))$ é não vazio, resta verificar que $\forall\theta \in \mathcal{U}(\hat{\sigma}'_k(s), \hat{\sigma}'_k(t))$, $\theta = \theta \circ \sigma_{k+1}$. De fato, sabemos que σ_{k+1} é a ligação obtida no $(k+1)$-ésimo passo do algoritmo de unificação e, portanto temos as seguintes possibilidades exclusivas para σ_{k+1}, onde a primeira possibilidade representa o caso em que $\hat{\sigma}'_k(s) = \hat{\sigma}'_k(t)$ e as duas últimas possibilidades representam o caso em que $\hat{\sigma}'_k(s) \neq \hat{\sigma}'_k(t)$.

i) $\sigma_{k+1} = id$: Neste caso é trivialmente verdade que $\theta = \theta \circ \sigma_{k+1}$, pois $\theta = \theta \circ id$ qualquer que seja a substituição θ, em particular para $\theta \in \mathcal{U}(\hat{\sigma}'_k(s), \hat{\sigma}'_k(t))$ temos que o resultado também é válido.

ii) $\sigma_{k+1} = \{(\hat{\sigma}'_k(s))|_p / (\hat{\sigma}'_k(t))_p\}$: Neste caso temos que $(\hat{\sigma}'_k(s))|_p$ é uma variável que não ocorre em $(\hat{\sigma}'_k(t))|_p$, e p é a posição onde ocorre a primeira diferença mais externa e mais à esquerda entre os termos $\hat{\sigma}'_k(s)$ e $\hat{\sigma}'_k(t)$. Assim, para uma variável y qualquer vamos mostrar que $\theta(y) = (\theta \circ \sigma_{k+1})(y)$. Analogamente à análise de casos feita na base de indução, temos:

$y = (\hat{\sigma}'_k(s))|_p$**:** Neste caso, temos que:

$$\begin{aligned}
\theta(y) &= \theta((\hat{\sigma}'_k(s))|_p) \\
&= \hat{\theta}((\hat{\sigma}'_k(s))|_p), \text{ pela observação da definição de substituição} \\
&= \hat{\theta}((\hat{\sigma}'_k(t))|_p), \text{ pois } \theta \in \mathcal{U}(\hat{\sigma}'_k(s), \hat{\sigma}'_k(t)) \\
&= \hat{\theta}(\sigma_{k+1}((\hat{\sigma}'_k(s))|_p)), \text{ pois } \sigma_{k+1}((\hat{\sigma}'_k(s))|_p) = (\hat{\sigma}'_k(s))|_p \\
&= (\theta \circ \sigma_{k+1})((\hat{\sigma}'_k(s))|_p), \text{ por definição} \\
&= (\theta \circ \sigma_{k+1})(y).
\end{aligned}$$

$y \neq (\hat{\sigma}'_k(s))|_p$**:** Neste caso, temos diretamente que $\theta(y) = (\theta \circ \sigma_{k+1})(y)$. Pois, como y é diferente de $(\hat{\sigma}'_k(s))|_p$, temos que $y \notin Dom(\sigma_{k+1})$, o que implica que $\sigma_{k+1}(y) = y$.

iii) $\sigma_{k+1} = \{(\hat{\sigma}'_k(t))|_p / (\hat{\sigma}'_k(s))_p\}$: Este caso é inteiramente análogo ao caso (ii) anterior e portanto não o repetiremos aqui.

Assim, concluímos que os termos $\hat{\sigma}'_k(s)$ e $\hat{\sigma}'_k(t)$ são unificáveis e $\forall\theta \in \mathcal{U}(\hat{\sigma}'_k(s), \hat{\sigma}'_k(t))$ vale que $\theta = \theta \circ \sigma_{k+1}$, para todo natural k. Logo, segue a demonstração do lema. ∎

Note que no Lema 2.2.14, demonstramos que os termos gerados por instanciação dos termos s e t iniciais, pela substituição iterada σ'_{k-1}, computada até o $(k-1)$-ésimo passo, são ainda unificáveis pela mesma substituição θ unificador de s e t. Esta demonstração difere da geralmente encontrada em livros texto e envolve uma discussão sobre o domínio de θ.

Lema 2.2.15: (Terminação) *Sejam s e t dois termos unificáveis, tais que $s \neq t$, e σ'_{k-1}, para $k > 0$, a substituição iterada, obtida na $(k-1)$-ésima iteração do algoritmo de unificação, tendo como entrada dois termos unificáveis s e t. Assim, se σ_k é a ligação obtida na k-ésima iteração do algoritmo e se $\hat{\sigma}'_{k-1}(s) \neq \hat{\sigma}'_{k-1}(t)$, então*

$$Card(\ \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(s))) \cup \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(t)))\ )$$
$$< Card(\ \mathcal{V}ars(\hat{\sigma}'_{k-1}(s)) \cup \mathcal{V}ars(\hat{\sigma}'_{k-1}(t))\ ).$$

Demonstração: De fato, sabemos que se $\hat{\sigma}'_{k-1}(s) \neq \hat{\sigma}'_{k-1}(t)$, então na iteração seguinte o algoritmo de unificação busca a posição p onde ocorre a primeira diferença entre os termos $\hat{\sigma}'_{k-1}(s)$ e $\hat{\sigma}'_{k-1}(t)$, e a partir desta posição computa a ligação σ_k, para a qual temos duas opções:

a) $\sigma_k = \{(\hat{\sigma}'_{k-1}(s))|_p/(\hat{\sigma}'_{k-1}(t))|_p\}$, quando $(\hat{\sigma}'_{k-1}(s))|_p$ é uma variável que não ocorre em $(\hat{\sigma}'_{k-1}(t))|_p$.

b) $\sigma_k = \{(\hat{\sigma}'_{k-1}(t))|_p/(\hat{\sigma}'_{k-1}(s))|_p\}$, quando $(\hat{\sigma}'_{k-1}(t))|_p$ é uma variável que não ocorre em $(\hat{\sigma}'_{k-1}(s))|_p$.

Em ambos os casos a análise é a mesma, portanto vamos considerar apenas um caso. Assim, suponha que σ_k seja como no item (a). Como $(\hat{\sigma}'_{k-1}(s))|_p$ é uma variável que não ocorre em $(\hat{\sigma}'_{k-1}(t))|_p$, temos que, denotando $(\hat{\sigma}'_{k-1}(s))|_p$ por x e $(\hat{\sigma}'_{k-1}(t))|_p$ por r:

$$x \notin \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(s))) \text{ e } x \notin \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(t)))$$
$$\Rightarrow x \notin \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(s))) \cup \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(t))),$$

pois σ_k substitui todas as ocorrências de x por r. Logo, os termos instanciados por σ_k não possuem a variável x. Por outro lado, temos que:

$$x \in \mathcal{V}ars(\hat{\sigma}'_{k-1}(s)) \cup \mathcal{V}ars(\hat{\sigma}'_{k-1}(t)),$$

pois x é um subtermo do termo $\hat{\sigma}'_{k-1}(s)$.

Além disso, como r é um subtermo do termo $\hat{\sigma}'_{k-1}(t)$, temos que σ_k não introduz no conjunto $\mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(s))) \cup \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(t)))$, variáveis distintas das que ocorriam no conjunto $\mathcal{V}ars(\hat{\sigma}'_{k-1}(s)) \cup \mathcal{V}ars(\hat{\sigma}'_{k-1}(t))$. Assim,

$$Card(\ \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(s))) \cup \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(t)))\ )$$

$$= Card(\ \mathcal{V}ars(\hat{\sigma}'_{k-1}(s)) \cup \mathcal{V}ars(\hat{\sigma}'_{k-1}(t))\ ) - 1$$

$$\Downarrow$$

$$Card(\ \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(s))) \cup \mathcal{V}ars(\hat{\sigma}_k(\hat{\sigma}'_{k-1}(t)))\ )$$

$$< Card(\ \mathcal{V}ars(\hat{\sigma}'_{k-1}(s)) \cup \mathcal{V}ars(\hat{\sigma}'_{k-1}(t))\ ).$$

Daí segue a demonstração do lema. ■

Como já comentamos anteriormente, na linha 08 do algoritmo verificamos se o subtermo de s na posição p é uma variável e se ocorre no subtermo de t na posição p. E caso a resposta seja negativa, passamos à linha 10, onde fazemos esta mesma verificação para o termo t em vez de s. Isto nos garante que quando realizarmos a ligação $\{x/r\}$, o termo r não possui ocorrências da variável x. Esta verificação é importante, pois garante a correção do algoritmo, contudo torna-o ineficiente, pois pode ser que seja necessário gerar termos exponencialmente grandes durante o processo de unificação.

Nos teoremas seguintes, verificamos que o algoritmo de unificação é correto e completo, isto é, para quaisquer dois termos s e t o algoritmo dá uma resposta, caso s e t não sejam unificáveis o algoritmo falha, mas caso s e t sejam unificáveis o algoritmo computa uma substituição que é de fato um *mgu* de s e t.

Teorema 2.2.16: (Correção) *Sejam s e t dois termos unificáveis. Se σ é a substituição computada pelo algoritmo de unificação, então $\sigma \in \mathcal{U}(s,t)$.*

Demonstração: Vamos verificar este fato por indução em $n = Card(\mathcal{V}ars(s) \cup \mathcal{V}ars(t))$.

B.I.: Para $n = 0$, temos que $\mathcal{V}ars(s) = \emptyset$ e $\mathcal{V}ars(t) = \emptyset$. Portanto os termos s e t são termos funcionais, sem ocorrências de variáveis, que são unificáveis. Logo, devemos

ter $s = t$, mas neste caso a substituição computada pelo algoritmo de unificação é a substituição id. E é trivialmente verdade que $id(s) = id(t)$ para $s = t$. Portanto $\sigma = id \in \mathcal{U}(s,t)$.

P.I.: Suponha que a hipótese seja válida para $n-1$ e vamos mostar que também será para n. O caso em que $s = t$ é trivial e o argumento é o mesmo da base de indução, portanto vamos nos ater ao caso em que $s \neq t$. Pelo Lema 2.2.15 temos que se σ_1 é a substituição computada pelo algoritmo de unificação em um passo de execução, então

$$Card(\ \mathcal{V}ars(\hat{\sigma}_1(s))\ \cup\ \mathcal{V}ars(\hat{\sigma}_1(t))\ ) = Card(\ \mathcal{V}ars(s)\ \cup\ \mathcal{V}ars(t)\ )\ -1.$$

Assim, supondo que a substituição computada pelo algoritmo de unificação para os termos $\hat{\sigma}_1(s)$ e $\hat{\sigma}_1(t)$, que pelo Lema 2.2.14 são unificáveis, é um elemento do conjunto $\mathcal{U}(\hat{\sigma}_1(s), \hat{\sigma}_1(t))$, vamos mostrar que o mesmo vale para os termos s e t. Note que, se $\theta = \textit{Robinson-Unification}(\hat{\sigma}_1(s), \hat{\sigma}_1(t))$, e se $\sigma = \textit{Robinson-Unification}(s,t)$, então $\sigma = \theta \circ \sigma_1$. De fato, esta é a composição retornada pelo algoritmo de unificação. Assim,

$$\begin{aligned}
\theta \in \mathcal{U}(\hat{\sigma}_1(s), \hat{\sigma}_1(t)) \quad &\Rightarrow \quad \hat{\theta}(\hat{\sigma}_1(s)) = \hat{\theta}(\hat{\sigma}_1(t)) \\
&\Rightarrow \quad (\widehat{\theta \circ \sigma_1})(s) = (\widehat{\theta \circ \sigma_1})(t) \\
&\Rightarrow \quad \hat{\sigma}(s) = \hat{\sigma}(t) \\
&\Rightarrow \quad \sigma \in \mathcal{U}(s,t).
\end{aligned}$$

Com isto concluímos a demonstração do teorema. ∎

Acabamos de verificar que a substituição computada pelo algoritmo de unificação para dois termos unificáveis s e t é de fato um unificador de s e t. No teorema seguinte verificamos que esta substituição é mais geral que qualquer outra substituição que unifique os termos s e t.

Teorema 2.2.17: (Generalidade) *Sejam s e t dois termos unificáveis. Se σ é a substituição computada pelo algoritmo de unificação, então para toda substituição $\theta \in \mathcal{U}(s,t)$ temos que $\sigma \lesssim \theta$.*

Demonstração: Novamente vamos usar indução em $n = Card(\mathcal{V}ars(s) \cup \mathcal{V}ars(t))$ para verificar este fato. Note que queremos mostrar que $\forall \theta \in \mathcal{U}(s,t),\ \exists \delta$ tal que $\theta = \delta \circ \sigma$.

B.I.: Pelo mesmo argumento do lema anterior, temos que para $n = 0$ os termos s e t devem ser iguais. Neste caso $\sigma = id$. Assim, $\forall \theta \in \mathcal{U}(s,t),\ \theta = \theta \circ \sigma$. Logo, σ é mais geral que θ.

P.I.: Suponha que a hipótese seja válida para $n - 1$ e vamos mostrar que também será para n. O caso em que $s = t$ é trivial e o argumento é o mesmo da base de indução. Portanto vamos nos ater ao caso em que $s \neq t$. Pelo Lema 2.2.15 temos que se σ_1 é a substituição computada pelo algoritmo de unificação em um passo de execução, então

$$Card(\ \mathcal{V}ars(\hat{\sigma}_1(s))\ \cup\ \mathcal{V}ars(\hat{\sigma}_1(t))\ ) = Card(\ \mathcal{V}ars(s)\ \cup\ \mathcal{V}ars(t)\ )\ - 1.$$

Assim, supondo que a substituição computada pelo algoritmo de unificação para os termos $\hat{\sigma}_1(s)$ e $\hat{\sigma}_1(t)$, que pelo Lema 2.2.14 são unificáveis, é mais geral que qualquer elemento do conjunto $\mathcal{U}(\hat{\sigma}_1(s), \hat{\sigma}_1(t))$, vamos mostrar que o mesmo vale para os termos s e t.

Primeiro observamos que, pelo Lema 2.2.14, temos:

$$\forall \theta\ \in\ \mathcal{U}(s,t),\ \ \exists \alpha \text{ tal que } \theta = \alpha \circ \sigma_1.$$

Afirmamos que $\alpha \in \mathcal{U}(\hat{\sigma}_1(s), \hat{\sigma}_1(t))$. De fato,

$$\begin{aligned} \theta \in \mathcal{U}(s,t) &\Rightarrow \hat{\theta}(s) = \hat{\theta}(t) \\ &\Rightarrow (\widehat{\alpha \circ \sigma_1})(s) = (\widehat{\alpha \circ \sigma_1})(t) \\ &\Rightarrow \hat{\alpha}(\hat{\sigma}_1(s)) = \hat{\alpha}(\hat{\sigma}_1(t)) \\ &\Rightarrow \alpha \in \mathcal{U}(\hat{\sigma}_1(s), \hat{\sigma}_1(t)). \end{aligned}$$

Observe ainda que se $\sigma_1' = \textit{Robinson-Unification}(\hat{\sigma}_1(s), \hat{\sigma}_1(t))$, e se $\sigma = \textit{Robinson-Unification}(s,t)$, então $\sigma = \sigma_1' \circ \sigma_1$. De fato, esta é a composição computada pelo algoritmo de unificação.

Assim, temos que para $\sigma'_1 = \textit{Robinson-Unification}(\hat{\sigma}_1(s), \hat{\sigma}_1(t))$, existe uma substituição δ tal que $\alpha = \delta \circ \sigma'_1$, por hipótese de indução. Logo,

$$\begin{aligned} \theta = \alpha \circ \sigma_1 \ &\Rightarrow \ \theta = \delta \circ \sigma'_1 \circ \sigma_1 \\ &\Rightarrow \ \theta = \delta \circ \sigma. \end{aligned}$$

Ou seja, para todo $\theta \in \mathcal{U}(s,t)$, existe δ tal que $\theta = \delta \circ \sigma$, onde σ é a substituição computada pelo algoritmo de unificação para os termos s e t. Logo, temos que σ é mais geral que θ seja qual for $\theta \in \mathcal{U}(s,t)$. Em outras palavras $\sigma \lesssim \theta$. ■

Teorema 2.2.18: (Completude) *Sejam s e t dois termos quaisquer. Se s e t são unificáveis, então o algoritmo de unificação finaliza apresentando um unificador mais geral de s e t. Caso s e t não sejam unificáveis o algoritmo de unificação para reportando tal fato.*

Demonstração: Para verificar a completude do algoritmo de unificação, devemos analisar o caso em que os termos do parâmetro de entrada não são unificáveis, e verificar que neste caso o processo de unificação falha e o caso em que os termos são unificáveis, e verificar que neste caso o algoritmo computa um *mgu*.

No caso em que s e t não são unificáveis, nenhum σ gerado pelo algoritmo será solução de $s \stackrel{?}{=} t$. Portanto o algoritmo deve parar reportanto tal fato. Isto se dá nas linhas 06, 08 e 10 do algoritmo.

- Na linha 06 do algoritmo, verificamos se na posição onde ocorre a primeira diferença entre os termos s e t, os dois subtermos $s|_p$ e $t|_p$ não são variáveis. Caso obtenhamos uma resposta positiva para esta verificação, temos que nesta posição ocorrem apenas símbolos de função nos dois termos. E como trata-se da posição onde ocorre a primeira diferença, concluímos que estes símbolos de função são sintaticamente distintos. Logo, os termos s e t não são unificáveis e o algoritmo para reportando tal fato.

- Na linha 08 do algoritmo verificamos, caso o subtermo $s|_p$ seja uma variável, se $s|_p$ ocorre no subtermo $t|_p$. Mas caso tenhamos uma resposta positiva para esta verificação, então $s|_p$ é uma variável diferente do termo $t|_p$ e, além disso $s|_p \in$

$\mathcal{V}ars(t|_p)$. Logo, segue do Lema 2.2.13 que a equação $s|_p \stackrel{?}{=} t|_p$ não tem solução. Então o algoritmo para e reporta que s e t não são unificáveis.

- Na linha 10 do algoritmo faz-se a mesma verificação realizada na linha 05, porém agora para saber se $t|_p \in \mathcal{V}ars(s|_p)$, caso $t|_p$ seja uma variável. Então da mesma forma, se a resposta para esta verificação é positiva, o algoritmo para reportando que os termos s e t não são unificáveis.

Caso os termos s e t sejam unificáveis, a substituição gerada pelo algoritmo de unificação é um *mgu* de s e t. De fato, devemos demonstrar que se σ é a substituição gerada pelo algoritmo, então σ é uma solução da equação $s \stackrel{?}{=} t$ e para toda substituição α que também seja unificador de s e t existe uma substituição δ tal que $\alpha = \delta \circ \sigma$. Mas este fato segue diretamente dos teoremas 2.2.16 e 2.2.17. ∎

Capítulo 3

Semântica do PVS

O PVS[1] (*Prototype Verification System*) é um sistema de especificação e verificação que provê um ambiente integrado para desenvolvimento e análise de especificações formais, e suporta uma ampla gama de atividades envolvendo criação, análise, modificação, gerenciamento e documentação de teorias e provas. O PVS é basicamente composto por uma *Linguagem de Especificação* fortemente integrada com um poderoso *Assistente de Prova Interativo*, um *Typechecker*, além de outras ferramentas como *bibliotecas de especificação*, dentre outras. Neste capítulo apresentamos uma visão geral sobre a semântica do PVS e as principais ferramentas utilizadas na especificação que apresentamos no Capítulo 4. Para mais detalhes sobre a semântica do PVS ver [19], o qual é a base das seções 3.3 e 3.4 deste capítulo. Para mais detalhes sobre a linguagem de especificação do PVS, o provador e o sistema em si, veja os manuais [21,22,30], os quais formam a base das seções 3.1 e 3.2. Para mais detalhes sobre a *teoria* `trs`, brevemente apresentada na seção 3.5, ver [8].

3.1 A Linguagem de Especificação do PVS

A *linguagem de especificação* do PVS é baseada em lógica de ordem superior simplesmente tipada, e é desenvolvida para permitir especificações suscintas e legíveis, além de construções efetivas de prova. Dentro de uma *teoria* os tipos podem ser definidos a partir de tipos mais básicos, como booleanos, números naturais, etc.

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

Exemplo 3.1.1: Por exemplo, para definir uma substituição idempotente foi construída a especificação do tipo `idempotent_sub`, apresentada abaixo. Tal especificação consta da *sub-teoria* `substitution`, e foi definida a partir do tipo `boll` e também a partir do tipo `Sub`, pois `sigma` é um objeto previamente declarado como sendo de tipo `Sub`, o que significa que `sigma` é uma substituição.

```
idempotent_sub?(sigma): bool = comp(sigma, sigma) = sigma
idempotent_sub: TYPE = (idempotent_sub?)
```

Uma especificação em PVS consiste de uma coleção de *teorias*. E cada *teoria* consiste de um conjunto de símbolos para os nomes dos tipos e constantes introduzidas na *teoria*, além de axiomas, definições e teoremas associados. Como exemplo tomamos a especificaçao da *sub-teoria* `unification`, apresentada no Apêndice A.

3.2 O Assistente de Provas

O objetivo principal do assistente de prova do PVS é dar suporte à construção de provas legíveis, isto é, o processo de verificação busca ser próximo de uma prova que faríamos no papel, logo permite interação humana, de forma que a prova possa ser facilmente entendida e comunicada a outras pessoas. No sentido de fazer com que as provas sejam facilmente desenvolvidas, o assistente de prova do PVS provê uma coleção poderosa de comandos de prova, que quando combinados corretamente desenvolvem uma estratégia de prova que visa realizar raciocínios lógicos, proposicionais e aritméticos, com o uso de definições e lemas.

O assistente de prova do PVS foi desenvolvido com base na semântica usual de Gentzen da Teoria da Prova. Isto significa que os objetivos em PVS são apresentados como sequentes $\Gamma \vdash \Delta$, onde Γ e Δ são sequências finitas de fórmulas.

O assistente de prova mantém uma árvore de prova, e o objetivo é obter uma árvore de prova que seja completa, isto é, uma árvore em que todos as folhas são reconhecidas como verdadeiras. Cada nó da árvore é um objetivo de prova, representado por um sequente

da forma $\Gamma \vdash \Delta$ que consiste de uma sequência de fórmulas Γ chamadas antecedentes e uma sequência de fórmulas Δ chamadas consequentes. A interpretação de um sequente é que a conjunção dos antecedentes implica a disjunção dos consequentes, isto é, para $\Gamma = \{A_1, A_2, A_3, \ldots\}$ e $\Delta = \{B_1, B_2, B_3, \ldots\}$, temos que $(A_1 \wedge A_2 \wedge A_3 \ldots) \supset (B_1 \vee B_2 \vee B_3 \ldots)$.

3.3 A Checagem de Tipos em PVS

Sabemos que os tipos formam um mecanismo poderoso para detectar erros sintáticos e semânticos, isto é feito através de uma operação de checagem de tipos, que em PVS é mantida pelo *typechecker*. A lógica expressiva do PVS proporciona uma boa integração entre o *typechecker* e o *assistente de prova*. O *typechecker*, explora o poder dedutivo do *assistente de prova* para provar automaticamente as condições de correção de tipos, ou TCC's do inglês *type correctness conditions*, que são obrigações de prova geradas pela operação de checagem de tipos. Estas obrigações de prova, surgem por exemplo quando realizamos a checagem de tipos de um termo em confronto com um subtipo de um predicado. Tais obrigações também aparecem como sub-objetivos de prova durante uma formalização.

Em PVS os tipos básicos consistem de *Booleanos*, `bool`, e *números reais*, `real`. O PVS é uma linguagem de especificação fortemente tipada, onde os tipos são construídos a partir dos tipos básicos, através de funções e produtos de tipos, e expressões são construídas a partir das constantes e variáveis por meio de aplicações, abstrações e sequências.

A operação de checagem de tipos é feita dentro de um *contexto*. Em PVS, um contexto Γ é uma sequência de declarações, onde cada declaração é ou uma declaração de tipo, `s:TYPE`, ou uma declaração de constante, `c:T`, onde `T` é um tipo, ou uma declaração de variável, `x:VAR T`. Isto pode ser visto como uma função parcial que associa a cada símbolo uma *espécie* que pode ser ou `TYPE`, ou `CONSTANT`, ou `VARIABLE` e um *tipo* a cada símbolo de constante e variável. De maneira mais formal, definimos:

Definição 3.3.1: Seja Γ um contexto e s um símbolo com declaração D e r um símbolo qualquer. Então,

1. $(\Gamma, s : D)(s) = D$

2. $r \neq s \Rightarrow (\Gamma, s : D)(r) = \Gamma(r)$

3. Se s não é declarado em Γ, então $\Gamma(s)$ é indefinido.

4. Para qualquer símbolo, s a espécie de s em Γ é dada por $kind(\Gamma(s))$.

5. Se $kind(\Gamma(s))$ é `CONSTANT` ou `VARIABLE`, então o tipo de $\Gamma(s)$ é o tipo associado a s em Γ.

As regras de tipos em uma teoria simplesmente tipada são dadas pela definição recursiva de uma função parcial τ que associa:

(i) a um termo a, bem tipado em relação a um contexto Γ, um tipo $\tau(\Gamma)(a)$.

(ii) a um tipo A, bem formado em relação a um contexto Γ, a palavra-chave `TYPE` como resultado de $\tau(\Gamma)(A)$.

(iii) a um contexto Δ, bem formado em relação a um contexto Γ, a palavra-chave `CONTEXT` como resultado de $\tau(\Gamma)(\Delta)$.

Normalmente, as regras de tipos são apresentadas como regras de inferência, mas em PVS uma apresentação funcional é mais apropriada, pois desta forma obtemos uma argumentação de correção de provas mais natural e direta. Assim temos a seguinte definição para as regras de tipo, que em PVS representam a operação de *typechecking*.

Definição 3.3.2: Regras de tipos

$$
\begin{array}{rcl}
\tau()(\{\}) & = & \texttt{CONTEXT} \\
\tau()(\Gamma, s : \texttt{TYPE}) & = & \texttt{CONTEXT},\ \text{se } \Gamma(s) \text{ é indefinido } \ \text{e } \tau()(\Gamma) = \texttt{CONTEXT} \\
\tau()(\Gamma, c : T) & = & \texttt{CONTEXT},\ \text{se } \Gamma(c) \text{ é indefinido, } \tau(\Gamma)(T) = \texttt{TYPE} \\
 & & \text{e } \tau()(\Gamma) = \texttt{CONTEXT} \\
\tau()(\Gamma, x : \texttt{VAR}\ T) & = & \texttt{CONTEXT},\ \text{se } \Gamma(x) \text{ é indefinido, } \tau(\Gamma)(T) = \texttt{TYPE} \\
 & & \text{e } \tau()(\Gamma) = \texttt{CONTEXT} \\
\tau(\Gamma)(s) & = & \texttt{TYPE},\ \text{se } kind(\Gamma(s)) = \texttt{TYPE} \\
\tau(\Gamma)([A \rightarrow B]) & = & \texttt{TYPE},\ \text{se } \tau(\Gamma)(A) = \tau(\Gamma)(B) = \texttt{TYPE} \\
\tau(\Gamma)([A_1, A_2]) & = & \texttt{TYPE},\ \text{se } \tau(\Gamma)(A_i) = \texttt{TYPE} \text{ para } 1 \leq i \leq 2
\end{array}
$$

$$\begin{aligned}
\tau(\Gamma)(s) &= \mathit{type}(\Gamma(s)),\ \text{se } \mathit{kind}(\Gamma(s)) \in \{\texttt{CONSTANT}, \texttt{VARIABLE}\}\\
\tau(\Gamma)(f\ a) &= B,\ \text{se } \tau(\Gamma)(f) = [A \to B] \text{ e } \tau(\Gamma)(a) = A\\
\tau(\Gamma)(\lambda(x : T) : a) &= [T \to \tau(\Gamma, x : \texttt{VAR}\ T)(a)],\ \text{se } \Gamma(x) \text{ é indefinido}\\
&\quad \text{e } \tau(\Gamma)(T) = \texttt{TYPE}\\
\tau(\Gamma)((a_1, a_2)) &= [\tau(\Gamma)(a_1), \tau(\Gamma)(a_2)]\\
\tau(\Gamma)(p_i\ a) &= T_i,\ \text{onde } \tau(\Gamma)(a) = [T_1, T_2]
\end{aligned}$$

Exemplo 3.3.1: Seja Ω um contexto onde `bool` : `TYPE`, `TRUE` : `bool` e `FALSE` : `bool`. Assim, neste contexto as regras de tipo são dadas por:

$$\begin{aligned}
\tau()(\{\}) &= \texttt{TYPE}\\
\tau()(\Omega) &= \texttt{TYPE}\\
\tau(\Omega)([[\texttt{bool}, \texttt{bool}] \to \texttt{bool}]) &= \texttt{TYPE}\\
\tau(\Omega)((\texttt{TRUE}, \texttt{FALSE})) &= [\texttt{bool}, \texttt{bool}]\\
\tau(\Omega)(p_2(\texttt{TRUE}, \texttt{FALSE})) &= \texttt{bool}\\
\tau(\Omega)(\lambda(x : \texttt{boll}) : \texttt{TRUE}) &= [\texttt{bool} \to \texttt{bool}]
\end{aligned}$$

Note que nas regras de tipos a boa formação do contexto em questão não é explicitamente verificada, contudo tais regras preservam a boa formação do contexto em cada chamada recursiva, então se o contexto inicial é bem formado, todos os contextos intermediários o serão também.

Um termo bem tipado s com um tipo designado por τ dentro de um contexto Γ é dito um termo de tipo $\tau(\Gamma)(s)$ no contexto Γ.

Expomos no Capítulo 4 a apresentação da formalização de uma teoria para unificação de primeira ordem. Obviamente esta teoria passou pela operação de checagem de tipos, que é feita explicitamente através do comando `M-x type-check-prove` ou `M-x tcp`, mas também é realizada automaticamente pelo PVS durante uma formalização. Assim, alguns TCC's foram gerados e mostraremos abaixo um deles como exemplo.

Exemplo 3.3.2: Na Tabela 4.2.1 apresentamos um dos três construtores, denominado

`resolving_diff`, especificado com o objetivo de obter uma função que compute unificadores mais gerais. O resultado da checagem de tipos deste construtor gera sete TCC's, um deles é apresentado aqui para exemplificar a importância da checagem de tipos e como obtemos resultados com o procedimento realizado pelo *typechecking* representado pelas regras da Definição 3.3.2. Na Seção 4.2, apresentamos este construtor detalhadamente, mas para entender este exemplo precisamos saber que este construtor age recursivamente em dois parâmetros de tipo termo que são diferentes e especificados como tipos dependentes unificáveis. Assim, com a operação de checagem de tipos esperamos, durante o processo de recursão, aplicar este construtor a termos que ainda sejam unificáveis e diferentes, isto é, preserva a tipagem. Portanto, o *typechecker* gera uma obrigação de prova, um TCC, apresentado pelo PVS como segue:

```
 Subtype TCC generated (at line 100, column 57) for
     subtermOF(t,  #(k + 1))
     expected type  {t: term |
                            unifiable(subtermOF(s,  #(1 + k)), t) &
                             NOT subtermOF(s,  #(1 + k)) = t}
   proved - complete

resolving_diff_TCC4: OBLIGATION
  FORALL (s: term, (t: term | unifiable(s, t) & s /= t), f: symbol,
          st: {args: finite_sequence[term] | args`length = arity(f)}):
    NOT st`length = 0 AND s = app(f, st) IMPLIES
     (FORALL (fp: symbol,
              stp:{args: finite_sequence[term] | args`length = arity(fp)}):
        t = app(fp, stp) IMPLIES
         (FORALL (k: below[stp`length]):
            k = min({kk: below[stp`length] |
                    subtermOF(s,  #(kk + 1)) /= subtermOF(t,  #(kk + 1))})
             IMPLIES
             unifiable(subtermOF(s,  #(1 + k)), subtermOF(t,  #(k + 1)))
              &
              NOT subtermOF(s,  #(1 + k)) = subtermOF(t,  #(k + 1))));
```

Este TCC foi gerado para o termo `subtermOF(t, #(k + 1))` e o tipo esperado para este termo é que ele seja unificável com o termo `subtermOF(s, #(k + 1))` e diferente deste, esta informação vem descrita no "cabeçalho" do TCC. Logo no corpo do TCC, onde temos a obrigação de prova a ser demonstrada, isto é, sob as hipóteses de que `s` e `t` sejam dois termos unificáveis e diferentes, onde o termo `s` é uma aplicação, devemos ter que o termo `subtermOF(t, #(k + 1))` é do tipo esperado. Esta verificação se faz necessária devido ao

caráter recursivo do construtor `resolving_diff`.

O PVS ainda oferece o recurso de declarar tipos dependentes, isto é, alguns dos tipos dos parâmetros de uma especificação podem ser dependentes de parâmetros anteriores. Este recurso foi importante na especificação proposta neste trabalho, pois nos permitiu trabalhar com termos unificáveis, como observamos no exemplo anterior, onde temos que sobre o termo `t` existe a condição `unifiable(s, t)`, que indica que sobre o tipo do termo `t` existe a condição de que `t` seja unificável com o termo `s`.

3.4 As Regras de Prova do PVS

As regras de prova do PVS são apresentadas em termos de cálculo de sequentes. Como mencionamos na Seção 3.2, um sequente é da forma $\Sigma \vdash_\Gamma \Lambda$, onde Γ é o contexto, Σ é o conjunto das fórmulas que compõem o antecedente e Λ representa o conjunto das fórmulas que compõem o consequente. Sobre um sequente desta forma devemos fazer a seguinte leitura: a conjunção das fórmulas de Σ implica a disjunção das fórmulas de Λ.

3.4.1 Regras Estruturais

Com as regras estruturais podemos rearranjar um sequente ou enfraquecê-lo, introduzindo novas fórmulas na conclusão.

A regra (W) apresentada em 3.4.1, representa uma poderosa regra de enfraquecimento. Todas as regras estruturais podem ser expressas em termos desta. Esta regra permite derivar um sequente mais fraco de um mais forte, por meio de introdução de fórmulas no antecedente ou no consequente. Este fato é expresso pela condição que se impõe de que $\Sigma_1 \subseteq \Sigma_2$ e $\Lambda_1 \subseteq \Lambda_2$.

$$\frac{\Sigma_1 \vdash_\Gamma \Lambda_1}{\Sigma_2 \vdash_\Gamma \Lambda_2}\,(W), \quad \text{se } \Sigma_1 \subseteq \Sigma_2 \text{ e } \Lambda_1 \subseteq \Lambda_2 \tag{3.4.1}$$

As regras de contração $(C \vdash)$ e $(\vdash C)$, apresentadas abaixo, permitem substituir várias ocorrências de uma mesma fórmula no antecedente ou no consequente por uma

única ocorrência.

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

As regras de comutação $(X \vdash)$ e $(\vdash X)$, afirmam que a ordem das fórmulas tanto do antecedente quanto do consequente é insignificante.

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

3.4.2 Regra de Corte

A regra de corte (Cut) pode ser usada para introduzir a análise de casos sobre uma fórmula a, dentro de uma prova de um sequente da forma $\Sigma \vdash_\Gamma \Lambda$. Isto leva a dois sub-objetivos da forma $\Sigma, a \vdash_\Gamma \Lambda$ e $\Sigma \vdash_\Gamma a, \Lambda$, que podem ser vistos como assumindo a em um ramo da prova e $\neg a$ no outro ramo da prova.

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

3.4.3 Regras para Axiomas Proposicionais

Na regra (Ax), simplesmente temos afirmação trivial de que a segue de a.

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

As regras $(\texttt{FALSE} \vdash)$ e $(\vdash \texttt{TRUE})$, afirmam que se em um dado sequente existe uma ocorrência de `FALSE` no antecedente ou uma ocorrência de `TRUE` no consequente, então este sequente é um axioma.

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

3.4.4 Regras de Contexto

Algumas fórmulas valem em um contexto simplesmente porque elas já fazem parte do contexto, ou como uma fórmula mesmo ou como uma declaração de constante. Nisto consiste a assertiva das regras $(ContextFormula)$ e $(ContextDefinition)$ abaixo.

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

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

Um contexto Γ pode ser estendido, através das regras $(Context \vdash)$ e $(\vdash Context)$, por fórmulas no antecedente ou negação de fórmulas no consequente.

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

A regra $(ContextW)$, é uma regra de enfraquecimento do contexto que é bastante útil, pois mostra que uma derivação é monótona em relação a um contexto.

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

3.4.5 Regras Condicionais

As regras $(IF \vdash)$ e $(\vdash IF)$ têm o objetivo de eliminar as ocorrências de `IF-THEN-ELSE` em uma prova. Contudo estas regras não são usuais pois elas aumentam o contexto.

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

3.4.6 Regras de Igualdade

As regras de transitividade e simetria para a igualdade podem ser derivadas das regras de igualdade ($Refl$) e ($Repl$) abaixo. Nestas regras a notação $a[e]$ indica uma ou mais ocorrências de e na fórmula a, tal que não existem ocorrências de variáveis livres em e. Similarmente, a notação $\Lambda[e]$ indica ocorrências de e em Λ.

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

3.4.7 Regras de Igualdade Booleana

Na regra ($Repl$ TRUE) temos a asserção de que uma fórmula a no antecedente pode ser tratada como uma fórmula de igualdade no antecedente da forma $a = \texttt{TRUE}$. Similarmente a regra ($Repl$ FALSE) asserta que uma fórmula a no consequente pode ser vista como uma igualdade da forma $a = \texttt{FALSE}$, no antecedente. Já a regra TRUE-FALSE, afirma que as constantes booleanas TRUE e FALSE são distintas.

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

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

3.4.8 Regras de Redução

As regras de redução (β) e (π), são axiomas de igualdade, que nos possibilitam realizar simplificações óbvias, para aplicações envolvendo lambda abstrações e projeção de produtos.

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

3.4.9 Regras de Extensionalidade

As regras de extensionalidade (*FunExt*) e (*TupExt*), são regras que estabelecem igualdade para expressões funcionais e de produto respectivamente. A regra de extensionalidade para funções introduz uma constante de Skolem s para estabelecer que duas funções f e g são iguais quando os resultados das aplicações destas funções a um argumento qualquer de s são iguais. A regra de extensionalidade para produto nos diz que dois produtos são iguais se as suas projeções correspondentes são iguais.

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

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

3.4.10 Regra de Restrição de Tipo

A regra de restrição de tipo (*Typepred*), é usada para suprir a necessidade de uma regra que introduza a restrição de tipo sobre um termo como uma fórmula no antecedente de um dado sequente.

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

Exemplo 3.4.1: Vejamos por exemplo a prova do teorema `Vars_is_var`, da *sub_teoria* `subterm`, onde formalizamos o fato de que o conjunto de variáveis de um termo que é uma variável é igual ao próprio termo. Nesta prova o primeiro sequente é apresentado da seguinte forma:

```
    |-------
{1}   FORALL (t: term[variable, symbol, arity]):
        vars?(t) => Vars(t) = ({y: (V) | y = t})
```

A este sequente aplicamos a regra de prova `skeep`, a fim de skolemizar as variáveis ligadas. Em seguida decompomos a igualdade de conjuntos presente na fórmula, com a regra `decompose-equality` e com isto temos um novo sequente:

```
[-1]  vars?(t)
  |-------
{1}   Vars(t)(x!1) = (x!1 = t)
```

Neste ponto a estratégia adotada é a de converter a igualdade da fórmula 1 em uma dupla implicação com a regra `iff`. Em seguida expandimos a definição de `Vars` e realizamos uma simplificação proposicional, o que faz com que a prova se divida em dois sub-objetivos, o primeiro deles dado por:

```
{-1}  EXISTS (p: positions?(t)): subtermOF(t, p) = x!1
[-2]  vars?(t)
  |-------
{1}   x!1 = t
```

No objetivo acima, pela fórmula -1 que existe uma posição `p`, pertencente ao conjunto de posições do termo `t`, tal que o subtermo de `t` nesta posição é igual à variável `x!1`. Mas como o termo `t` é uma variável, temos que esta posição é a raiz, o que implica que `t` é igual `x!1`. Assim, skolemizando a variável da fómula -1 e expandindo algumas definições, chegamos a esta conclusão. E isto completa este ramo da prova, pois a fómula 1 representa a negação do fato de `t` ser igual `x!1`.

No ramo seguinte da prova, temos a outra parte da dupla-implicação, representada pelo seguinte sequente:

```
{-1}  x!1 = t
[-2]  vars?(t)
  |-------
{1}   EXISTS (p: positions?(t)): subtermOF(t, p) = x!1
```

Note que agora, sabendo que `t` é igual `x!1`, pela fórmula -1, e que `t` é uma variável, pela fórmula -2, queremos verificar que existe uma posição `p`, pertencente ao conjunto de posições do termo `t`, tal que o subtermo de `t` nesta posição é igual à `x!1`. Mas pela fórmula 1, temos que tal posição não existe. Então a estratégia é instanciar a fórmula 1 com a posição raiz. Com isto novamente completamos este ramo da prova, após expandir a definição de `subtermOF`. Em seguida provamos em um outro objetivo que a posição raiz faz parte do conjunto de posições do termo `t` e com isto completamos a formalização deste lema.

3.5 *Sub-teorias* da *Teoria* TRS

Nesta seção apresentamos algumas *sub-teorias* da *teoria* `trs`, onde encontramos as especificações e formalizações dos conceitos iniciais definidos no Capítulo 2, como por exemplo as definições de termos, posições, substituições, etc. A especificação destes conceitos e a formalização de vários resultados acerca dos mesmos, compõe o que chamamos de uma álgebra de termos e formam base para o desenvolvimento da *sub-teoria* `unification`.

3.5.1 A *Sub-teoria* term

Como vimos na Definição 2.2.2, termos são definidos recursivamente. Contudo o PVS não permite definições de tipos recursivos, pois uma definição ou declaração de tipo em um contexto deve conter somente símbolos previamente declarados no contexto. Para contornar esta situação o PVS oferece o mecanismo chamado `DATATYPE`, que é uma forma de definição de tipo recursivo. Para mais informações sobre o mecanismo *abstract datatype* do PVS veja [18]. Assim, a forma que se encontrou de especificar a noção de termos em PVS foi utilizando a ferramenta `DATATYPE`.

A especificação de termos é apresentada na *sub-teoria* `term` como abaixo:

```
term[variable: TYPE+, symbol: TYPE+, arity: [symbol -> nat]] : DATATYPE
 BEGIN

 vars(v: variable): vars?

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

 END term
```

Note que os termos são definidos como sendo uma variável, ou uma aplicação de um símbolo `f` a um argumento `args` composto por uma sequência finita de termos, sendo que o símbolo `f` tem aridade igual ao comprimento da sequência de termos. Veja que a chamada recursiva do `DATATYPE` é promovida por `args`.

3.5.2 A *Sub-teoria* `positions`

A *sub-teoria* `positions` contém a especificação da noção de conjunto de posições de um termo e formalizações de algumas propriedades envolvendo posições.

```
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 <<
```

Note que no construtor `positionsOF`, obtemos um tipo `positions`, que é um conjunto de posições. Por sua vez, uma posição é definida como um tipo `finseq[posnat]`, isto é, uma sequência finita de naturais. O conjunto de posições de um termo, obtido pelo construtor `positionsOF`, é especificado conforme a Definição 2.2.3.

Dentre os lemas formalizados na *sub-teoria* `positions`, os mais importantes para o desenvolvimento da *sub-teoria* `unification` e que foram mais diretamente usados nas formalizações apresentadas em `unification` foram os lemas `positions_of_terms_finite` e `positions_of_arg`, cuja especificação apresentamos abaixo:

```
positions_of_terms_finite : LEMMA is_finite(positionsOF(t))

positions_of_arg : LEMMA
     FORALL ( (s : term | app?(s) ), k : below[length(args(s))] ) :
     positionsOF(s)( #[posnat]( k+1 ) )
```

O primeiro estabelece que o conjunto de posições de um termo é finito. O segundo estabelece que se um termo `s` é uma aplicação e se `k` é um natural menor que o comprimento da sequência de argumentos do termo `s`, então `k+1` pertence ao conjunto de posições do termo `s`.

3.5.3 A *Sub-teoria* `subterm`

Na *sub-teoria* `subterm` temos a especificação da definição de subtermo, encontrada na Definição 2.2.4.

```
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)
```

A especificação do construtor `subtermOF` é a principal desta teoria, contudo encontramos a especificação de outra definição importante, a de conjunto de variáveis de um termo `t`, denotado por `Vars(t)`.

Temos também na *teoria* `subterm` a formalização de vários resultados acerca de termos e subtermos que foram largamente utilizados nas formalizações da *sub-teoria* `unification`. Por exemplo, temos a formalização dos lemas `vars_of_term_finite` e `term_eq_subterm`, cuja especificação apresentamos abaixo:

```
vars_of_term_finite: LEMMA  is_finite(Vars(t))


term_eq_subterm : LEMMA
   positionsOF(s)(p) AND subtermOF(s, p) = s IMPLIES p = empty_seq
```

O primeiro estabelece que o conjunto de variáveis de um termo é finito. Este resultado é certo, pois como vimos os termos são definidos como sendo finitos. O segundo estabelece que se `p` pertence ao conjunto de posições de um termo `s` e se o subtermo de `s` nesta posição é o próprio `s`, então a posição `p` é a posição raiz ou, como tratado na especificação, é uma sequência vazia.

3.5.4 A *Sub-teoria* `substitution`

Na *sub-teoria* `substitution` encontramos as especificações que, de certa forma, são as mais importantes para o desenvolvimento da *sub-teoria* `unification`. Temos nesta subteoria a especificação das noções de substituição, domínio e imagem de uma substituição, conforme apresentado na Definição 2.2.6, além da definição de conjunto de variáveis da imagem de uma substituição.

```
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)}

VRan(sig): set[(V)] = IUnion(LAMBDA (x | Dom(sig)(x)): Vars(sig(x)))

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

Observamos que `sig` é uma variável declarada no preâmbulo da *sub-teoria* `substitution` como sendo do tipo `[(V) -> term]`, isto é, `sig` é uma aplicação de um conjunto de variáveis em um conjunto de termos, sem nenhuma restrição sobre estes conjuntos. Assim, a forma escolhida para definir substituição, foi definir um novo tipo `Sub`, que corresponde a responder *verdadeiro* ou *falso* para a questão de o domínio de `sig` ser finito.

Temos ainda na *sub-teoria* `substitution` a especificação da definição de extensão homeomórfica de uma substituição a um conjunto de termos. Tal especificação foi de grande importância para o desenvolvimento da *sub-teoria* `unification`.

```
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 <<
```

Encontramos ainda nesta teoria a formalização de vários resultados sobre substituições que foram largamente aplicados durante as formalizações da *sub-teoria* `unification`. Por exemplo, temos alguns resultados importantes sobre composição de substituições:

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

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

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

O lema `subs_o` estabelece que a composição de substituições é ainda uma substituição. O lema `ext_o` estabelece que a extensão homeomórfica de uma composição de duas substituições é igual à composição das extensões homeomórficas das substituições. O lema `o_ass` estabelece que a composição de substituições é associativa.

Capítulo 4

Formalização da Teoria de Unificação

Neste capítulo vamos apresentar a especificação da *teoria* `unification`, que construímos buscando formalizar o teorema de existência de unificadores mais gerais em teorias de primeira ordem. Para tanto, especificamos uma função composta de três construtores para representar a versão do algoritmo de unificação de Robinson apresentada no Capítulo 2, restrita a termos unificáveis. Na Seção 4.1, apresentamos a estrutura hierárquica da *teoria* `unification`. Na Seção 4.2, vamos explicar a organização desta *teoria* e os principais construtores especificados. Na Seção 4.3, apresentamos de maneira detalhada a formalização do teorema que fala sobre a existência de *mgu*'s, que é baseado no fato de que o algoritmo de unificação computa unificadores mais gerais. Na Seção 4.4 apresentamos a formalização dos lemas onde afirmamos que a substituição, computada na especificação proposta, é de fato um *mgu* de dois termos unificáveis. Na Seção 4.5, apresentamos a formalização de lemas importantes, onde temos garantida a terminalidade do algoritmo de unificação, e que foram fundamentais para a formalização dos lemas da seção anterior. Com estes teoremas provamos a completude da versão do algoritmo de unificação de Robinson.

4.1 Estrutura Hierárquica da Teoria `unification`

Nesta seção vamos apresentar a hierarquia da teoria `unification`, isto é, veremos sobre quais *teorias* a teoria `unification` foi construída. Na Figura 4.1.1, mostramos esta estrutura hierárquica, onde é possível observar quais *sub-teorias* da *teoria* `trs` foram importadas pela *sub-teoria* `unification`, sendo que apenas a *sub-teoria* `substitution` é importada diretamente. A *teoria* `ars` [7], representa um conjunto de *sub-teorias* para

Sistemas Abstratos de Redução, a hierarquia desta teoria pode ser observada em [8]. As *teorias* identity, finite_sets e finite_sequences, são as *teorias* para definir a função identidade, conjuntos finitos e sequências finitas, respectivamante, importadas do prelúdio do PVS. Para mais informações sobre a biblioteca do prelúdio do PVS ver [20].

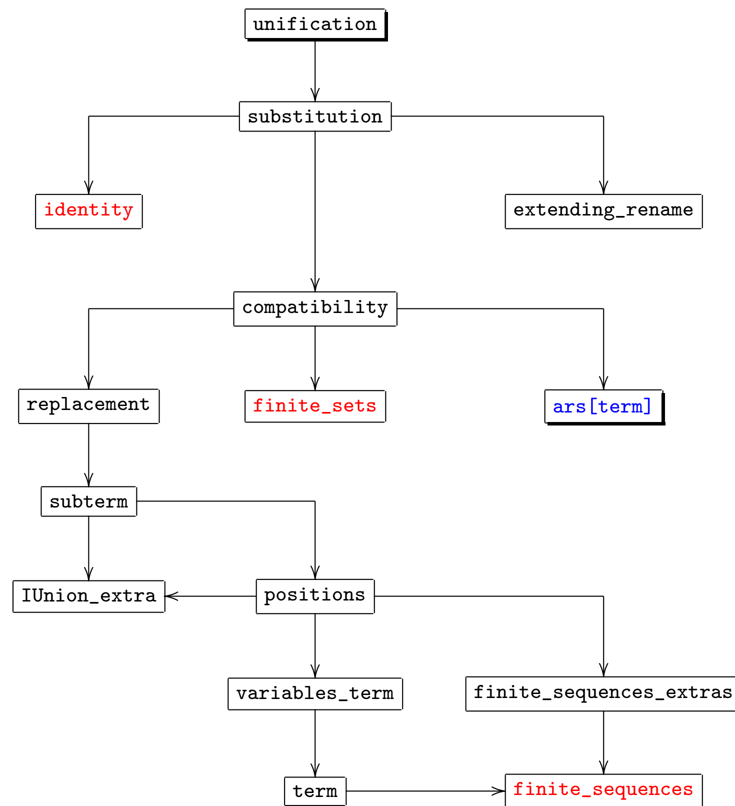


Figura 4.1.1: Estrutura hierárquica da *sub-teoria* unification

4.2 Organização da Teoria unification

A fim de especificar uma função que compute unificadores mais gerais para dois termos unificáveis, foram definidos separadamente três construtores. Observe que estamos considerando termos unificáveis, isto é, não consideramos para esta especificação o caso de falha no processo de unificação. Fazemos isto porque queremos demonstrar especificamente que existe um *mgu*, computado via algoritmo de unificação, para dois termos unificáveis quaisquer, que é o resultado necessário na formalização do teorema dos pares críticos de Knuth-Bendix [6]. Apresentamos a seguir a descrição destes construtores bem como a especificação de cada um. Depois apresentaremos brevemente as seções da *teoria* unification onde formalizamos várias propriedades destes construtores.

Tabela 4.2.1: Especificação do construtor `resolving_diff`

```
resolving_diff(s : term, (t : term | unifiable(s,t) & s /= t ) ):
  RECURSIVE position =
   (CASES s OF
      vars(s) : empty_seq,
      app(f, st) :
      IF length(st) = 0 THEN empty_seq
      ELSE
       (CASES t OF
          vars(t) : empty_seq,
          app(fp, stp) :
          LET k : below[length(stp)] =
            min({kk : below[length(stp)] |
                 subtermOF(s,#(kk+1)) /= subtermOF(t,#(kk+1))}) IN
            add_first(k+1,
                resolving_diff(subtermOF(s,#(k+1)),subtermOF(t,#(k+1))))
        ENDCASES)
      ENDIF
    ENDCASES)
MEASURE s BY <<
```

resolving_diff: Na Tabela 4.2.1 temos a estrutura da especificação deste construtor, que toma como parâmetros dois termos unificáveis e diferentes s e t e retorna uma posição p, que é a posição onde ocorre a primeira diferença entre os termos s e t. Note que neste construtor os termos do argumento de entrada devem ser, não somente unificáveis, mas também diferentes, isto porque simplesmente não faz sentido buscar diferenças em termos iguais. Esta posição é obtida buscando-se a diferença mais a esquerda e mais externa entre os dois termos. Note que nesta posição, se os termos são unificáveis, devemos ter que $s|_p$ é uma variável ou $t|_p$ é uma variável. Este construtor age recursivamente e tem como medida a ordem bem-fundada $<<$, que é gerada automaticamente pela operação de checagem de tipos (*typechecking*) da *sub-teoria* `term`. Podemos interpretar esta medida como uma ordem de imersão de subtermos em termos, isto é, dado um termo finito s qualquer subtermo de s tem comprimento menor que s, o que significa que a cardinalidade do conjunto de posições de qualquer subtermo de s é menor que a cardinalidade do conjunto de posições de s, Definição 2.2.3. Isto torna a recursão finita, pois consideramos termos com um conjunto finito de posições, segundo o lema `positions_of_terms_finite` formalizado na *sub-teoria* `positions`.

Tabela 4.2.2: Especificação do construtor sub_of_frst_diff

```
sub_of_frst_diff(s:term, (t:term | unifiable(s,t) & s /= t )): Sub =
 LET k : position = resolving_diff(s,t) IN
    LET sp = subtermOF(s,k) , tp = subtermOF(t,k) IN
       IF vars?(sp)
       THEN (LAMBDA (x : (V)) : IF x = sp THEN tp ELSE x ENDIF)
       ELSE (LAMBDA (x : (V)) : IF x = tp THEN sp ELSE x ENDIF)
       ENDIF
```

sub_of_frst_diff: Na Tabela 4.2.2 apresentamos a especificação deste construtor, que também toma como argumentos dois termos unificavéis e diferentes s e t, e retorna uma substituição. O objetivo deste construtor é computar uma substituição que resolva a primeira diferença entre os termos s e t. Isto é feito a partir da posição computada pelo construtor `resolving_diff`. Portanto, se $p =$ `resolving_diff`(s, t) é a posição onde ocorre a primeira diferença mais a direita e mais externa entre os termos s e t e se $\sigma =$ `sub_of_frst_diff`(s, t), então $\hat{\sigma}(s|_p) = \hat{\sigma}(t|_p)$. Veja que os termos do argumento de entrada deste construtor também devem ser diferentes, já que partimos da posição encontrada pelo construtor anterior. A forma como obtemos esta substituição neste construtor é a seguinte: se p é a posição obtida em `resolving_diff`(s, t), fazemos uma verificação para saber se o subtermo $s|_p$ é uma variável, digamos x, e caso o seja retorna-se uma substituição que liga x ao subtermo $t|_p$, isto é, $\sigma := \{x/t|_p\}$. Caso contrário, devemos ter que o subtermo $t|_p$ é uma variável, digamos x, e neste caso a substituição é dada por $\sigma := \{x/s|_p\}$.

unification_algorithm: Apresentamos na Tabela 4.2.3 a especificação deste construtor, que toma como argumentos dois termos unificáveis s e t, e computa uma substituição σ, que é um *mgu* de s e t. Este fato está formalizado em dois lemas que apresentamos na Seção 4.4. Note que agora não exigimos que os termos sejam diferentes, isto porque neste construtor estamos interessados em considerar a substituição identidade, que é claramente a substituição que soluciona a equação $s \stackrel{?}{=} t$ quando temos $s = t$. Assim, no caso em que os termos s e t são iguais, `unification_algorithm` retorna a substituição identidade, caso s e t sejam diferentes,

Tabela 4.2.3: Especificação do construtor unification_algorithm

```
unification_algorithm(s : term, (t : term | unifiable(s,t))) :
   RECURSIVE Sub =
   IF s = t THEN identity
   ELSE LET sig = sub_of_frst_diff(s, t) IN
     comp( unification_algorithm((ext(sig))(s) , (ext(sig)(t))) , sig)
   ENDIF
MEASURE Card(union(Vars(s), Vars(t)))
```

unification_algorithm retorna uma substituição dada pela composição da substituição obtida em sub_of_frst_diff, digamos σ_1, com a substituição dada por mais um passo de unification_algorithm aplicado a $\hat{\sigma_1}(s)$ e $\hat{\sigma_1}(t)$. Portanto, se no k-ésimo passo o construtor retornar uma substituição σ tal que $\hat{\sigma}(s) = \hat{\sigma}(t)$, então devemos ter $\sigma = id \circ \sigma_k \circ \ldots \circ \sigma_1$, onde

$$\sigma_j = \texttt{unification_algorithm}(\hat{\sigma}_{j-1}(\ldots\hat{\sigma}_1(s)), \hat{\sigma}_{j-1}(\ldots\hat{\sigma}_1(t))),$$

para $j = 1, \ldots, k$. Isto significa que unification_algorithm age recursivamente e esta recursão é finita pois adotamos como medida a cardinalidade do conjunto $\mathcal{V}ars(s) \cup \mathcal{V}ars(t)$, que é um conjunto finito, pois o conjunto de variáveis de um termo finito é finito, e a cada passo eliminamos uma variável de um conjunto finito de variáveis. O fato de que para um termo s, $\mathcal{V}ars(s)$ é finito está formalizado no lema vars_of_term_finite da *sub-teoria* subterm.

Assim, com os três construtores apresentados acima, especificamos uma função para computar unificadores mais gerais de dois termos unificáveis. Mas é preciso verificar que a especificação destes construtores está correta e que realmente obtemos um *mgu* com o auxílio desta especificação.

Para verificar que não há falhas na especificação realizamos um procedimento de checagem de tipos, invocado pela rotina *type check prove*. Ao realizar este procedimento o PVS tenta verificar automaticamente obrigações de prova geradas pelo *typecheker*, contudo ocorrem casos em que estas obrigações de prova são geradas mas o provador do PVS é incapaz de verificá-las automaticamente, pois ocorrem situações em que uma estratégia de

prova mais especifica e direcionada faz-se necessária. Nestes casos fazemos a verificação de tais obrigações de prova, com o auxílio das ferramentas de prova que o PVS possui. Estas obrigações de prova, que também podem surgir como sub-objetivos de prova durante uma demonstração, são denominadas *type correctness conditions* (ou TCC), e surgem quando um termo passa por uma checagem de tipos confrontada com um subtipo esperado de predicado.

Para verificar que os três construtores juntos realmente computam um *mgu* de dois termos unificáveis foram formalizados vários teoremas auxiliares, onde provamos várias propriedades sobre cada um dos construtores. Estes teoremas foram organizados em três seções da *teoria* `unification` da seguinte forma:

Lemmas about "resolving_diff": Como o próprio nome da seção sugere, demonstramos aqui vários lemas sobre o construtor `resolving_diff`. Para demonstrar os lemas desta seção foram fundamentais os TCC's gerados pela checagem de tipos do construtor `resolving_diff`. Na Tabela 4.2.4, apresentamos a especificação destes lemas, e em seguida, comentamos brevemente em que consiste cada um deles. Observe que em todos os lemas desta seção temos como hipótese que os termos s e t são unificáveis e diferentes. Na descrição dos lemas feita a seguir, denotaremos a posição onde ocorre a primeira diferença entre os termo s e t por p, isto é, $p =$ `resolving_diff`(s, t).

`resol_diff_nonempty_implies_funct_terms`: Neste lema provamos que se p não é a posição raiz, então os dois termos são funcionais.

`resol_diff_to_rest_resol_diff`: Este é um lema construtivo, onde mostramos que se p não é a posição raiz e se $p = i \circ q$, então a primeira diferença entre os subtermos $s|_i$ e $t|_i$ está na posição q.

`position_s_resolving_diff` e `position_t_resolving_diff`: Nestes lemas provamos que $p \in \mathcal{P}os(s)$ e $p \in \mathcal{P}os(t)$, respectivamente. Note que precisamos destes dois lemas porque o termo t possui um tipo dependente em relação ao termo s.

`resolving_diff_has_diff_argument`: Neste lema provamos que $s|_p \neq t|_p$. Note que este lema é importante, pois no construtor `sub_of_frst_diff` precisamos deste

Tabela 4.2.4: Especificação dos lemas sobre o construtor resolving_diff

```
resol_diff_nonempty_implies_funct_terms : LEMMA
   FORALL (s: term, (t: term | unifiable(s, t) & s /= t)):
          resolving_diff(s,t) /= empty_seq IMPLIES
          (app?(s) AND app?(t))

resol_diff_to_rest_resol_diff : LEMMA
   FORALL (s: term, (t: term | unifiable(s, t) & s /= t)):
       LET rd =  resolving_diff(s,t) IN
           rd /= empty_seq IMPLIES
           resolving_diff(subtermOF(s,#(first(rd))),
                          subtermOF(t,#(first(rd)))) = rest(rd)

position_s_resolving_diff : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t, p : position):
       p = resolving_diff(s, t) IMPLIES positionsOF(s)(p);

position_t_resolving_diff : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t, p : position):
       p = resolving_diff(s, t) IMPLIES positionsOF(t)(p);

resolving_diff_has_diff_argument : LEMMA
   FORALL (s : term, t : term | unifiable(s,t) & s /= t,
           p : position | positionsOF(s)(p) & positionsOF(t)(p)):
       p = resolving_diff(s, t) IMPLIES
         subtermOF(s, p) /= subtermOF(t, p)

resolving_diff_has_unifiable_argument : LEMMA
   FORALL (s : term, t : term | unifiable(s,t) & s /= t,
           p : position | positionsOF(s)(p) & positionsOF(t)(p)):
       p = resolving_diff(s, t) IMPLIES
         unifiable(subtermOF(s, p), subtermOF(t, p))

resolving_diff_vars : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t,
   p : position | positionsOF(s)(p) & positionsOF(t)(p)):
       p = resolving_diff(s, t) IMPLIES
   vars?(subtermOF(s, p)) OR vars?(subtermOF(t, p))
```

resultado, que aliás surge como um TCC de sub_of_frst_diff.

resolving_diff_has_unifiable_argument: Neste lema provamos que $s|_p$ e $t|_p$ são unificáveis. Da mesma forma que o anterior, este lema também é importante, pois no construtor sub_of_frst_diff precisamos deste resultado, que também surge como um TCC de sub_of_frst_diff.

resolving_diff_vars: Neste lema provamos que ou $s|_p$ é uma variável, ou $t|_p$ é uma

Tabela 4.2.5: Especificação de alguns lemas sobre o construtor sub_of_frst_diff

```
dom_sub_of_frst_diff_is : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t, sig : Sub):
       sig = sub_of_frst_diff(s, t) AND p = resolving_diff(s, t)
       IMPLIES
          IF vars?(subtermOF(s, p))
            THEN Dom(sig) = singleton(subtermOF(s, p))
            ELSE Dom(sig) = singleton(subtermOF(t, p))
          ENDIF

var_sub_1stdiff_not_member_term : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t):
       LET sig = sub_of_frst_diff(s ,t) IN
          FORALL ( x | member(x,Dom(sig)), r | member(r,Ran(sig) )) :
              NOT member(x, Vars(r))

sub_of_frst_diff_remove_x : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t):
       LET sig = sub_of_frst_diff(s, t) IN
          Dom(sig)(x) IMPLIES
              (NOT member(x, Vars(ext(sig)(s)))) AND
              (NOT member(x, Vars(ext(sig)(t))))

vars_sub_of_frst_diff_s_is_subset_union : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t):
       LET sig = sub_of_frst_diff(s, t) IN
          subset?(Vars(ext(sig)(s)), union( Vars(s), Vars(t)))

vars_sub_of_frst_diff_t_is_subset_union : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t):
       LET sig = sub_of_frst_diff(s, t) IN
          subset?(Vars(ext(sig)(t)), union( Vars(s), Vars(t)))

union_vars_ext_sub_of_frst_diff : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t) :
       LET sig = sub_of_frst_diff(s, t) IN
          union(Vars(ext(sig)(s)), Vars(ext(sig)(t)))
          = difference(union( Vars(s), Vars(t)), Dom(sig))
```

variável. Trata-se de um resultado simples mas que precisamos formalizar.

Lemmas about "sub_of_frst_diff": Nesta seção, formalizamos lemas sobre o construtor `sub_of_frst_diff`. Para demonstrar os lemas desta seção foi preciso formalizar alguns lemas auxiliares dispostos na seção denominada `Auxiliary lemmas about substitutions and unifiers`, que encontram-se no Apêndice A. Apresentamos nas tabelas 4.2.5 e 4.2.6, a especificação dos lemas sobre o construtor `sub_of_frst_diff`.

Tabela 4.2.6: Especificação dos principais lemas sobre o construtor `sub_of_frst_diff`

```
sub_of_frst_diff_unifier_o : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t):
       member(rho, U(s, t)) IMPLIES
          LET sig = sub_of_frst_diff(s, t) IN
             EXISTS theta : rho = comp(theta, sig)

ext_sub_of_frst_diff_unifiable : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t):
       LET sig = sub_of_frst_diff(s, t) IN
          unifiable(ext(sig)(s), (ext(sig)(t)))

vars_ext_sub_of_frst_diff_decrease : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t):
       LET sig = sub_of_frst_diff(s, t) IN
          Card(union( Vars(ext(sig)(s)), Vars(ext(sig)(t))))
          < Card(union( Vars(s), Vars(t)))
```

Comentaremos aqui os lemas da Tabela 4.2.5, os lemas apresentados na Tabela 4.2.6 serão descritos e comentados mais detalhadamente na Seção 4.5. Porém, ressaltamos que os lemas da Tabela 4.2.6, foram particularmente interessantes de demonstrar e a formalização destes foi fundamental para concluírmos TCC's importantes sobre o construtor `unification_algorithm`. Novamente, todos os lemas da Tabela 4.2.5 têm como hipótese que os termos s e t são unificáveis e diferentes. Na descrição destes lemas, feita a seguir, σ indicará a substituição dada por `sub_of_frst_diff`(s, t) e p a posição dada por `resolving_diff`(s, t).

`dom_sub_of_frst_diff_is`: Neste lema formalizamos um resultado importante sobre o domínio de σ, afirmamos que além de ser unitário, exitem apenas duas possibilidades, ou $Dom(\sigma) = \{s|_p\}$ ou $Dom(\sigma) = \{t|_p\}$.

`var_sub_1stdiff_not_member_term`: Neste lema provamos que se uma variável x é tal que $x \in Dom(\sigma)$ então para todo termo r, da imagem de σ, vale que x não pertence a $\mathcal{V}ars(r)$.

`sub_of_frst_diff_remove_x`: Neste lema afirmamos que os termos $\hat{\sigma}(s)$ e $\hat{\sigma}(t)$ não possuem ocorrências da variável x, que pertence ao domínio de σ.

`vars_sub_of_frst_diff_s_is_subset_union`: Neste lema, provamos que o conjunto $\mathcal{V}ars(\hat{\sigma}(s))$ é subconjunto da união dos conjuntos $\mathcal{V}ars(s)$ e $\mathcal{V}ars(t)$.

Tabela 4.2.7: Especificação dos lemas sobre o construtor unification_algorithm

```
unification_algorithm_gives_unifier : LEMMA
   unifiable(s,t) IMPLIES member(unification_algorithm(s, t), U(s, t))

unification_algorithm_gives_mg_subs : LEMMA
   member(rho, U(s, t)) IMPLIES unification_algorithm(s, t) <= rho
```

vars_sub_of_frst_diff_t_is_subset_union: Neste lema formalizamos o mesmo resultado do lema anterior para o termo t, isto é, mostramos que $\mathcal{V}ars(\hat{\sigma}(t))$ é um subconjunto de $\mathcal{V}ars(s) \cup \mathcal{V}ars(t)$.

union_vars_ext_sub_of_frst_diff: Neste lema provamos o fato de que o conjunto $\mathcal{V}ars(\hat{\sigma}(s)) \cup \mathcal{V}ars(\hat{\sigma}(t))$ é exatamente igual ao conjunto $\mathcal{V}ars(s) \cup \mathcal{V}ars(t)$ menos uma variável, aquela do domínio de σ. Este lema juntamente com os dois anteriores foram de grande importância na formalização do lema denominado vars_ext_sub_of_frst_diff_decrease (Tabela 4.2.6), o qual garante a terminalidade do construtor unification_algorithm.

Lemmas about "unification_algorithm": Nesta seção formalizamos dois lemas sobre o construtor unification_algorithm, onde provamos que a substituição obtida em unification_algorithm(s, t), sendo s e t dois termos unificáveis, é de fato um *mgu* de s e t. E estes dois lemas são o ponto de partida para provar o último teorema que formalizamos, o que garante a existência de *mgu*'s para dois termos unificáveis e cuja formalização apresentamos e discutimos na Seção 4.3. Na Tabela 4.2.7, apresentamos a especificação desta seção. Na Seção 4.4 apresentamos a formalização destes lemas e aqui uma breve descrição de cada um.

unification_algorithm_gives_unifier: Neste provamos que a substituição dada por unification_algorithm(s, t) é um unificador de s e t.

unification_algorithm_gives_mg_subs: Neste provamos que a substituição dada por unification_algorithm(s, t) é mais geral que qualquer substituição que seja unificador de s e t.

Ainda temos na parte inicial da *teoria* `unification` especificações de conceitos importantes onde definimos elementos fundamentais em teoria de unificação. Tais definições são as de *instância de um termo*, *substituição mais geral*, *unificação entre dois termos*, *mgu* além de lemas auxiliares, onde demonstramos que dois termos, funcionais e unificáveis, têm o mesmo símbolo de função na posição raiz e o mesmo número de argumentos. É possível observar o código da especificação destes conceitos no Apêndice A. Além disso, foi necessário formalizar outros lemas que, por uma questão de organização, foram especificados nas *sub-teorias* `position`, `substitution` e `subterm`.

Nas seções seguintes discutiremos os lemas mais importantes desta especificação, apresentando as suas formalizações.

4.3 Formalização do Teorema Sobre a Existência de mgu's

Nesta seção vamos apresentar a formalização do teorema principal deste trabalho. O teorema `unification`, que afirma que quaisquer dois termos unificáveis possuem um unificador mais geral. A especificação deste teorema encontra-se na seção `Existence of a most general unifier` da teoria `unification`. Vejamos abaixo a especificação deste teorema.

```
unification : LEMMA
   unifiable(s,t) => EXISTS theta : mgu(theta)(s,t)
```

Em seguida vamos enunciar o teorema e apresentar a formalização de sua prova. Na Figura 4.3.1 mostramos a árvore da prova do teorema `unification`, onde é possível visualizar de mareira mais intuitiva a estrutura da demonstração do teorema. Na árvore o símbolo $\vdash$ representa o sequente que temos num dado momento da prova. As regras de prova adotadas são enunciadas ao longo das arestas da árvore, posicionadas depois de cada sequente que antecede a sua aplicação.

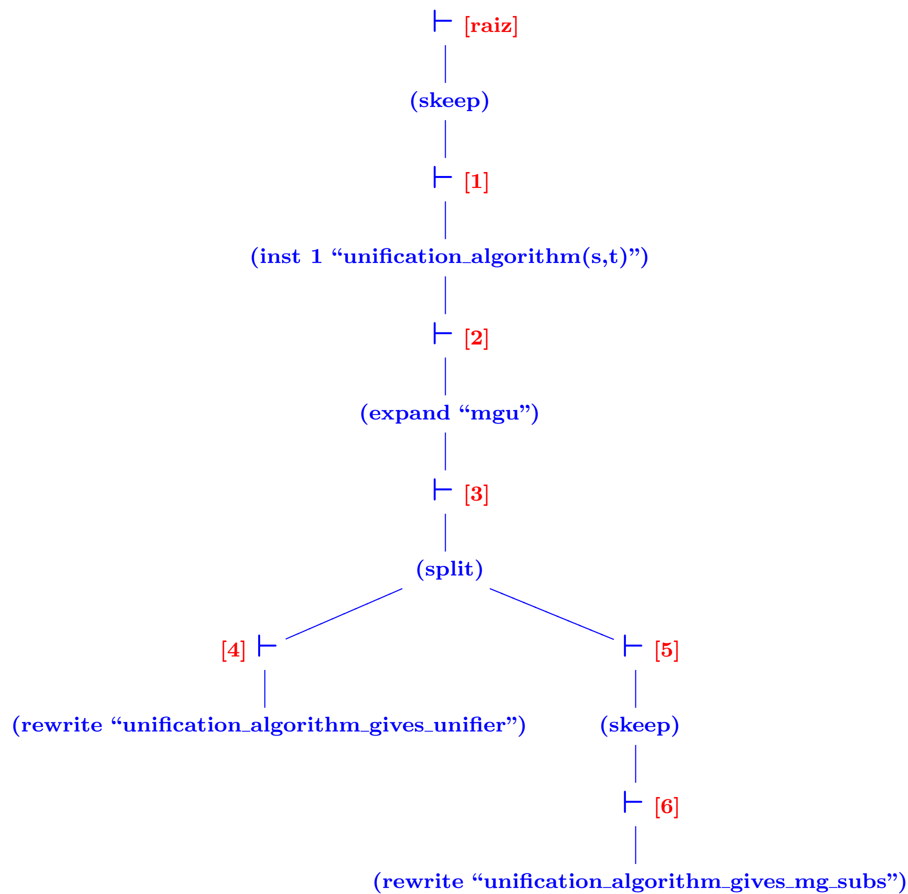


Figura 4.3.1: Árvore de prova do teorema `unification`.

Teorema 4.3.1: *Sejam s e t dois termos unificáveis, então existe uma substituição θ que é um* mgu *de s e t.*

Na demonstração deste teorema, utilizamos os lemas sobre o construtor `unification_algorithm`, intitulados `unification_algorithm_gives_unifier` e `unification_algorithm_gives_mg_subs`, cuja formalização apresentaremos na Seção 4.4. A utilização destes dois lemas torna a demonstração do Teorema 4.3.1 bastante simples. Isto porque no Teorema 4.3.1 mostramos apenas que dois termos unificáveis têm um *mgu*, que é obtido via *algoritmo de unificação*. O fato de que o algoritmo de unificação realmente computa um *mgu*, é mostrado nos dois lemas que utilizamos nesta prova. Então vamos apresentar em detalhes a prova formal do teorema `unification`.

Demonstração: Começamos a demonstração com o seguinte objetivo de prova,

- Sequente representado pelo nó [raiz],

```
  |-------
{1}   FORALL (s, t: term[variable, symbol, arity]):
        unifiable(s, t) => (EXISTS theta: mgu(theta)(s, t))
```

Este objetivo é o teorema que queremos demonstrar, ou seja, o objetivo principal que na árvore (Figura 4.3.1) de prova está representado pelo nó [raiz]. O primeiro comando de prova utilizado é o `skeep`, da biblioteca do PVS *Field*. Este comando é usado para skolemizar as variáveis das fórmulas quantificadas universalmente. Contudo `skeep` não introduz novos nomes para constantes, ele mantém os nomes das variáveis ligadas. Assim, depois de aplicar a regra `skeep`, obtivemos o seguinte sequente:

- Sequente representado pelo nó [1],

```
{-1}  unifiable(s, t)
  |-------
{1}   EXISTS theta: mgu(theta)(s, t)
```

A este sequente aplicamos a regra `(inst 1` "`unification_algorithm(s, t))`", que talvez seja a mais importante desta prova, pois fica claro como faremos a demonstração deste teorema, já que neste momento instanciamos a fórmula 1 com a substituição obtida pela aplicação de `unification_algorithm` aos termos s e t. Isto é, estamos lançando mão do fato, previamente demonstrado na teoria `unification`, mas que aqui será apresentado na Seção 4.4, de que `unification_algorithm` computa unificadores mais gerais. Daí temos o sequente:

- Sequente representado pelo nó [2],

```
[-1]  unifiable(s, t)
  |-------
{1}   mgu(unification_algorithm(s, t))(s, t)
```

Logo em seguida utilizamos a regra (`expand` "`mgu`"), onde expandimos a definição de `mgu`, o que nos direciona para a utilização dos lemas sobre o construtor `unification_algorithm`. E temos o sequente:

- Sequente representado pelo nó [3],

```
[-1]  unifiable(s, t)
  |-------
{1}   member(unification_algorithm(s, t), U(s, t)) &
        (FORALL tau: member(tau, U(s, t))
             IMPLIES unification_algorithm(s, t) <= tau)
```

Neste ponto, afim de dividir a prova da conjugação do sucedente, utilizamos o comando de prova `split`, que seleciona e divide uma fórmula conjuntiva, neste caso a fórmula selecionada é a fórmula 1 do sequente representado pelo nó [3].

Depois da aplicação da regra `split`, a árvore de prova se divide em dois ramos, isto significa que a partir deste ponto da prova temos dois sub-objetivos de prova. O primeiro onde temos que provar que a substituição dada por `unification_algorithm` é um unificador dos termos s e t e o segundo onde temos que provar que esta mesma substituição é mais geral do que qualquer outra substituição que seja um unificador dos termos s e t, mas como já mencionamos anteriormente, estes dois fatos foram previamente formalizados na *teoria* como resultados dos lemas que serão apresentados nas seções 4.4.1 e 4.4.2.

Assim, no sequente representado pelo nó [4], temos que provar que a substituição dada por `unification_algorithm(s, t)` é um unificador de s e t.

- Sequente representado pelo nó [4],

```
[-1]  unifiable(s, t)
  |-------
{1}   member(unification_algorithm(s, t), U(s, t))
```

Mas este é o resultado do lema `unification_algorithm_gives_unifier`. Portanto a regra de prova aplicada a este sequente é `(rewrite` "`unification_algorithm_gives_unifier)`", onde o comando de prova `rewrite` tenta determinar automaticamente as instanciações necessárias para combinar a conclusão do lema com as expressões das fórmulas que estamos tentando reescrever. Após este comando o provador encontra a devida instanciação e completa este ramo da prova, isto é, a folha deste ramo da árvore torna-se verdadeira.

Então, automaticamente passamos ao próximo objetivo da prova, expresso no sequente representado pelo nó [5], onde temos que provar que a substituição dada por `unification_algorithm(s, t)` é uma substituição que é mais geral do que qualquer outra substituição que seja um unificador dos termos s e t.

- Sequente representado pelo nó [5],

```
[-1]  unifiable(s, t)
  |-------
{1}   FORALL tau: member(tau, U(s, t))
          IMPLIES unification_algorithm(s, t) <= tau
```

Note que neste sequente a fórmula 1 é universalmente quantificada, portanto o primeiro comando de prova que aplicamos é `skeep`, o que nos leva ao último objetivo da prova, o sequente representado pelo nó [6].

- Sequente representado pelo nó [6].

```
{-1}  member(tau, U(s, t))
[-2]  unifiable(s, t)
  |-------
{1}   unification_algorithm(s, t) <= tau
```

Neste ponto, utilizamos novamente o comando de prova `rewrite`, aplicando o lema `unification_algorithm_gives_mg_subs`. Novamente o provador faz a instanciação necessária automaticamente e completa este ramo da prova. Com isto completamos a prova deste teorema.

Completar a árvore de prova significa concluir que todas as folhas da árvore são verdadeiras. Logo temos uma prova do objetivo principal da prova, representado pelo nó [raiz]. Portanto, mostramos que se s e t são dois termos unificáveis, então existe uma substituição que é um unificador mais geral de s e t. ∎

4.4 Formalização dos Lemas Sobre o Construtor `unification_algorithm`

Apresentamos a formalização dos lemas onde verificamos que a substituição computada pelo algoritmo de unificação é solução do dado problema de unificação e que é mais

geral. Tratam-se dos lemas aplicados diretamente na formalização do Teorema 4.3.1. Estes lemas correspondem aos teoremas sobre correção e generalidade do algoritmo de unificação apresentados na Seção 2.2.3.

4.4.1 Lema `unification_algorithm_gives_unifier`

Nesta seção apresentamos a formalização de um dos lemas sobre o construtor `unification_algorithm`, onde provamos que a substituição computada por este construtor aplicado a dois termos unificáveis, é um unificador destes dois temos. Na Tabela 4.2.7 temos a especificação deste lema. Este lema corresponde ao Teorema 2.2.16, onde provamos que o algoritmo de unificação é correto. Apresentamos a árvore de prova deste lema dividida nas figuras 4.4.1 e 4.4.2.

Lema 4.4.1: *Seja* $\sigma =$ `unification_algorithm`(s, t)*, onde* s *e* t *são dois termos unificáveis, então temos que* σ *é um unificador de* s *e* t*.*

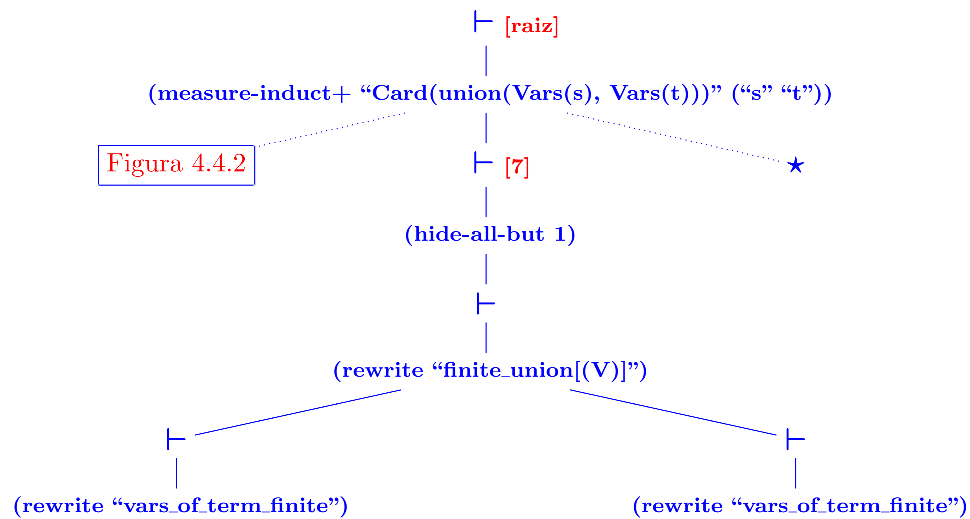


Figura 4.4.1: Início da árvore de prova do Lema 4.4.1. Os ramos denotados por ⋆, que são dois, são idênticos ao ramo abaixo do nó [7]. Tais ramos são gerados pela operação de checagem de tipos. O ramo principal da prova consta da Figura 4.4.2.

Demonstração: Assim, como na apresentação anterior, começamos expondo a árvore de prova gerada pelo PVS. Como já comentamos, dividimos a árvore de prova em duas figuras. Na Figura 4.4.1, mostramos o topo da árvore de prova, onde é possível visualizar o nó [raiz] e o início das ramificações obtidas pela aplicação da primeira regra utilizada. Na Figura 4.4.2, mostramos o ramo principal da árvore de prova, onde se dá a formalização

do lema propriamente. Os outros ramos, que podemos visualizar na Figura 4.4.1, são provenientes da checagem de tipos realizada pelo PVS.

Note que no construtor `unification_algorithm` utilizamos como medida a cardinalidade do conjunto dado pela união dos conjuntos de variáveis dos termos s e t, denotada por: `Card(union(Vars(s), Vars(t)))`. Portanto, a demonstração deste lema seguirá por indução nesta medida. Vamos expor a formalização deste lema apresentando alguns objetivos de prova, como são mostrados na interface do PVS, fazendo em seguida uma breve explicação do objetivo e das regras aplicadas a este. O objetivo inicial é apresentado da seguinte forma:

- Sequente representado pelo nó **[raiz]**.

```
  |-------
{1}   FORALL (s, t: term[variable, symbol, arity]):
        unifiable(s, t) IMPLIES
                member(unification_algorithm(s, t), U(s, t))
```

A este objetivo aplicamos a regra `measure-induct+` sobre a cardinalidade da união do conjunto de variáveis dos termos, que quer dizer que a nossa estratégia de prova é fazer indução na medida "`Card(union(Vars(s), Vars(t)))`". A aplicação desta regra nos leva a três sub-objetivos: o primeiro deles é o nosso objetivo principal, onde provamos por casos o lema propriamente dito; os dois outros sub-objetivos são resultado de uma checagem de tipos realizada pelo PVS, onde temos que provar que o conjunto `union(Vars(s), Vars(t))` é finito.

Assim, no primeiro destes sub-objetivos, provamos a hipótese do lema, que será feita por casos. Este sub-objetivo se apresenta da seguinte forma:

- Sequente representado pelo nó [1].

```
{-1}  FORALL (y_1: term[variable, symbol, arity]),
             (y_2: term[variable, symbol, arity]):
        Card(union(Vars(y_1), Vars(y_2))) <
         Card(union(Vars(x!1), Vars(x!2)))
         IMPLIES
         unifiable(y_1, y_2) IMPLIES
          member(unification_algorithm(y_1, y_2), U(y_1, y_2))
{-2}  unifiable(x!1, x!2)
  |-------
{1}   member(unification_algorithm(x!1, x!2), U(x!1, x!2))
```

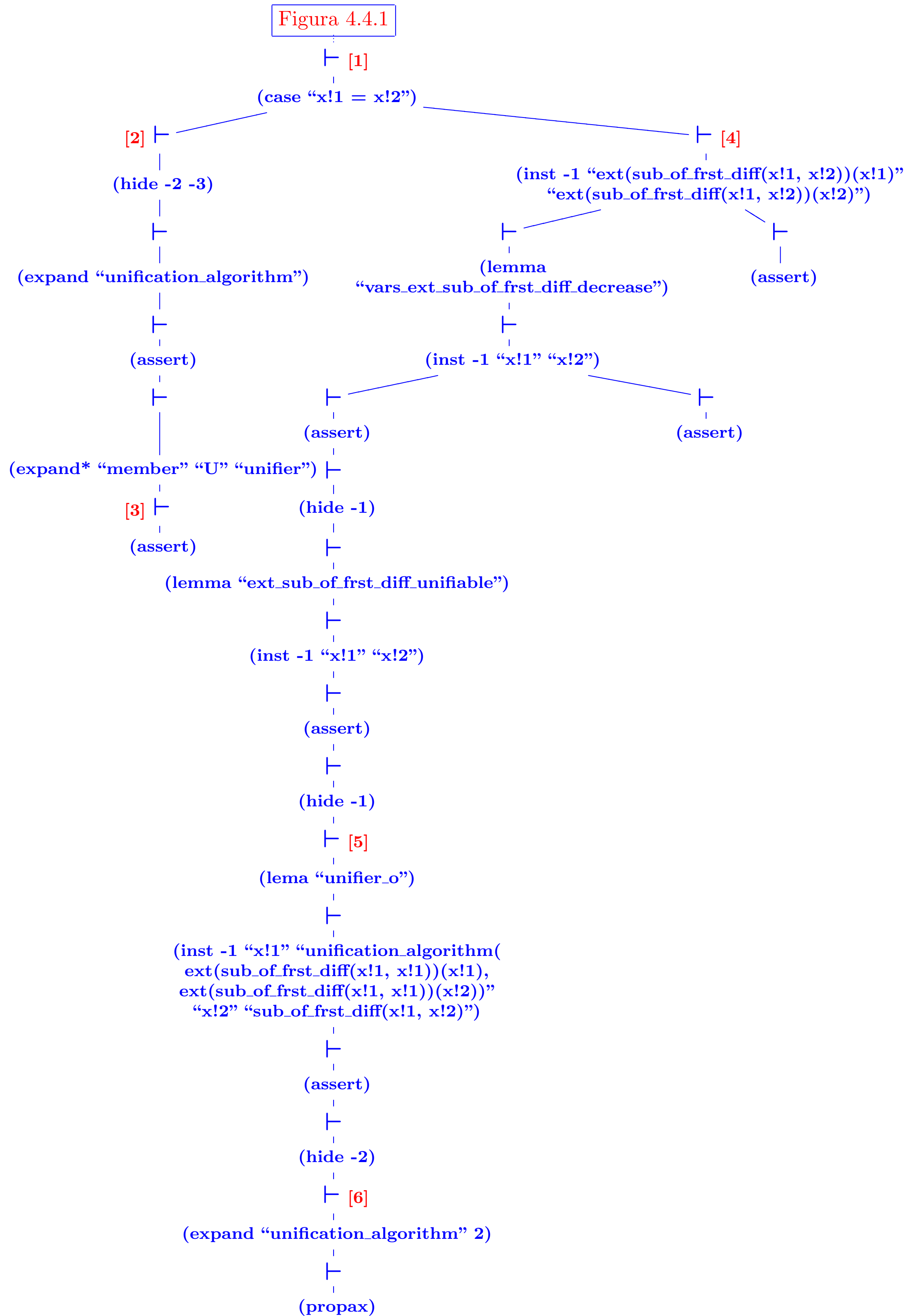


Figura 4.4.2: Ramo principal da árvore do prova do Lema 4.4.1. A parte inicial da árvore consta da Figura 4.4.1.

Neste sequente temos a hipótese de indução dada pelo antecedente -1, além da hipótese de que os termos `x!1` e `x!2` são unificáveis, e o nosso objetivo é provar o consequente 1. Neste ponto da prova aplicamos a regra "`case`" que nos leva a considerar dois casos. O primeiro caso onde "`x!1 = x!2`", e no segundo o caso onde "`x!1` $\neq$ `x!2`".

Vejamos rapidamente o primeiro caso, onde "`x!1 = x!2`", apresentado pelo provador do PVS na forma do seguinte sequente:

- Sequente representado pelo nó [2].

```
{-1}  x!1 = x!2
[-2]  FORALL (y_1: term[variable, symbol, arity]),
             (y_2: term[variable, symbol, arity]):
        Card(union(Vars(y_1), Vars(y_2))) <
         Card(union(Vars(x!1), Vars(x!2)))
         IMPLIES
         unifiable(y_1, y_2) IMPLIES
          member(unification_algorithm(y_1, y_2), U(y_1, y_2))
[-3]  unifiable(x!1, x!2)
  |-------
[1]   member(unification_algorithm(x!1, x!2), U(x!1, x!2))
```

Neste sequente começamos aplicando a regra `(hide -2 -3)`, onde escondemos as fórmulas -2 e -3, pois elas não serão necessárias na demonstração deste ramo da prova. Assim obtemos um sequente simplificado onde temos apenas as fórmulas -1 e 1. Neste ponto da prova utilizamos a estratégia de expandir as definições de "`member`", "`unification_algorithm`", "`U`" e "`unifier`", onde "`unifier`" aparece quando expandimos "`U`". Assim, obtemos o seguinte sequente:

- Sequente representado pelo nó [3]

```
[-1]  x!1 = x!2
  |-------
{1}   ext(identity)(x!1) = ext(identity)(x!2)
```

Se olharmos na especificação do construtor `unification_algorithm`, apresentada na Tabela 4.2.3, veremos que para `x!1 = x!2` este construtor retorna substituição identidade. Daí, obtemos na fórmula 1 a negação do fato de a substituição identidade pertencer ao conjunto dos unificadores de `x!1` e `x!2`. Mas isto nos leva a uma contradição, visto que na fórmula -1 temos que `x!1 = x!2`. Portanto com a regra `assert`, que emprega procedimentos de decisão para simplificar as fórmulas, completamos este ramo da prova.

Vejamos agora o caso em que "`x!1` $\neq$ `x!2`". Aqui começamos com o seguinte objetivo de prova:

- Sequente representado pelo nó [4]

```
[-1]  FORALL (y_1: term[variable, symbol, arity]),
             (y_2: term[variable, symbol, arity]):
        Card(union(Vars(y_1), Vars(y_2))) <
         Card(union(Vars(x!1), Vars(x!2)))
         IMPLIES
         unifiable(y_1, y_2) IMPLIES
          member(unification_algorithm(y_1, y_2), U(y_1, y_2))
[-2]  unifiable(x!1, x!2)
  |-------
{1}   x!1 = x!2
[2]   member(unification_algorithm(x!1, x!2), U(x!1, x!2))
```

Neste primeiro sequente deste ramo da prova, note que temos a hipótese que diferencia este caso do anterior na fórmula 1, que por estar no consequente representa a negação de "`x!1 = x!2`". Além disso, temos a hipótese de indução na fórmula -1. Neste ponto da prova, vamos instanciar a fórmula -1 com os termos "`ext(sub_of_frst_diff(x!1, x!2))(x!1)`" e "`ext(sub_of_frst_diff(x!1, x!2))(x!2)`". Assim, temos que se

```
Card(union( Vars(ext(sub_of_frst_diff(x!1, x!2))x!1),
            Vars(ext(sub_of_frst_diff(x!1, x!2))x!2) ))
< Card(union( Vars(x!1), Vars(x!2) ))
```

e se os termos

```
ext(sub_of_frst_diff(x!1, x!2))x!1 e ext(sub_of_frst_diff(x!1, x!2))x!1
```

são unificáveis, então a substituição dada por

```
unification_algorithm( ext(sub_of_frst_diff(x!1, x!2)),
                       ext(sub_of_frst_diff(x!1, x!2)) )
```

é um unificador destes termos.

Mas estes dois resultados estão previamente formalizados na *teoria*. Tratam-se dos lemas `vars_ext_sub_of_frst_diff_decrease` e `ext_sub_of_frst_diff_unifiable` da seção `Lemmas about` "`sub_of_frst_diff`" da *teoria* `unification`, cuja formalização apresentamos na Seção 4.5 e cuja especificação apresentamos na Tabela 4.2.6. Assim, após alguns passos de prova, onde fazemos a instanciação já mencionada e realizamos alguns procedimentos de simplificação, obtemos o seguinte sequente de prova:

- Sequente representado pelo nó [5].

```
[-1]  member(unification_algorithm(ext(sub_of_frst_diff(x!1, x!2))(x!1),
                                   ext(sub_of_frst_diff(x!1, x!2))(x!2)),
             U(ext(sub_of_frst_diff(x!1, x!2))(x!1),
               ext(sub_of_frst_diff(x!1, x!2))(x!2)))
[-2]  unifiable(x!1, x!2)
  |-------
[1]   x!1 = x!2
[2]   member(unification_algorithm(x!1, x!2), U(x!1, x!2))
```

Neste último sequente, temos que a fórmula -1 e a fórmula 2, são equivalentes. Apenas temos que nos dar conta do fato de que dadas duas substituições α e β, e dois termos unificáveis s e t, se α é um unificador de $\widehat{\beta}(s)$ e $\widehat{\beta}(t)$, então a composição $\alpha \circ \beta$ é um unificador de s e t. Este resultado está demonstrado no lema `unifier_o` da seção `Auxiliary lemmas about substitutions and unifiers`, cuja formalização apresentamos no Apêndice A, que vamos aplicar neste momento da prova. Assim, utilizando este lema, fazendo as devidas instanciações, e após alguns passos de prova para simplificar obtemos o seguinte:

- Sequente representado pelo nó [6]

```
[-1]  member(comp(unification_algorithm(
                        ext(sub_of_frst_diff(x!1, x!2))(x!1),
                        ext(sub_of_frst_diff(x!1, x!2))(x!2)),
                  sub_of_frst_diff(x!1, x!2)),
            U(x!1, x!2))
  |-------
[1]   x!1 = x!2
[2]   member(unification_algorithm(x!1, x!2), U(x!1, x!2))
```

Note que agora está clara a equivalência entre as fórmulas -1 e 2. Assim, neste ponto expandimos a definição de `unification_algorithm` e com isto completamos este ramo da prova, desde que `x!1` $\neq$ `x!2`.

Tendo completado a formalização dos dois ramos de prova gerados pela aplicação da regra `case` "`x!1 = x!2`", completamos o ramo principal da prova. Ressaltamos que nestes dois ramos da prova, omitimos alguns sub-objetivos de prova, que temos que provar principalmente quando usamos a estratégia de aplicar um outro lema da teoria, pois nestes casos o PVS gera sub-objetivos de prova onde temos que verificar asserções acerca do tipo das variáveis usadas para instanciar os lemas utilizados.

Nos dois ramos seguintes da prova, como já mencionamos anteriormente, temos que provar que os termos estão bem tipados. Temos dois objetivos de prova, onde "questiona-se" sobre a finitude da união dos conjuntos de variáveis dos termos, isto é, temos que provar que, dados dois termos s e t então $Vars(s) \cup Vars(t)$ é um conjunto finito.

A demonstração destes dois ramos, novamente segue pela aplicação de dois outros lemas, um da teoria `finite_sets` do prelúdio do PVS, denominado `finite_union`, que diz que a união de dois conjuntos finitos é ainda um conjunto finito. Mas ao aplicar este lema somos levados a provar que os conjuntos de variáveis dos dois termos em questão são conjuntos finitos, e isto segue de um outro lema da teoria `subterm` denominado `vars_of_term_finite`, que afirma que o conjunto de variáveis de um termo é finito. Assim, para ilustrar o que acabamos de explicar, apresentamos a seguir o objetivo de um destes ramos da prova. O outro ramo prova-se com a mesma sequência de comandos de prova. Temos o seguinte sequente inicial:

- Sequente representado pelo nó [7]

```
    |-------
{1}   is_finite[(V)]
          (union[((V))]
               (Vars[variable, symbol, arity](s!1),
                Vars[variable, symbol, arity](t!1)))
[2]   FORALL (s, t: term[variable, symbol, arity]):
        unifiable(s, t) IMPLIES
                member(unification_algorithm(s, t), U(s, t))
```

A este sequente aplicamos os lemas mencionados acima, realizamos os procedimentos de prova necessários para encontrar as devidas instanciações, realizamos as simplificações necessárias e com isto concluímos este ramo da prova.

Tendo completado a formalização dos ramos de prova gerados pela aplicação da regra `measure-induct+` "`Card(union(Vars(s), Vars(t)))`" ("`s`" "`t`"), concluímos a prova do lema. ∎

4.4.2 Lema `unification_algorithm_gives_mg_subs`

Nesta seção apresentamos a formalização do lema sobre o construtor `unification_algorithm`, onde provamos que a substituição computada por este construtor aplicado a dois termos unificáveis, é uma substituição mais geral que qualquer outra substituição que seja um unificador destes dois temos. Na Tabela 4.2.7 temos a especificação deste lema. Este lema corresponde ao Teorema 2.2.17, onde provamos a generalidade da substituição computada pelo algoritmo de unificação. Apresentamos a árvore de prova deste lema nas figuras 4.4.3, 4.4.4 e 4.4.5.

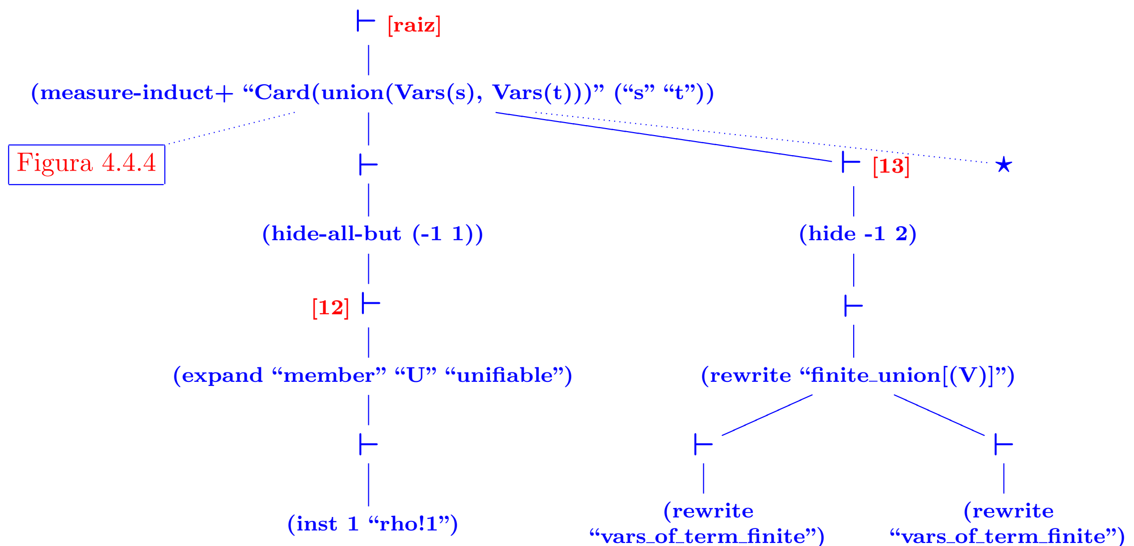


Figura 4.4.3: Início da árvore de prova do Lema 4.4.2. A ramificação denotada por ⋆ representa outros dez sub-objetivos de prova gerados pelo *typechecker*. Tais objetivos são provados ou da mesma forma que o ramo abaixo do nó [12] ou da mesma forma que o ramo abaixo do nó [13]. O ramo principal consta das figuras 4.4.4 e 4.4.5.

Lema 4.4.2: *Sejam* $\sigma = \texttt{unification_algorithm}(s,t)$ *e* $\theta \in \mathcal{U}(s,t)$*, onde* s *e* t *são dois termos unificáveis, então existe uma substituição* δ *tal que* $\theta = \delta \circ \sigma$*.*

Demonstração: Similarmente ao que foi feito na formalização do Lema 4.4.1, também neste lema a prova se dará por indução na medida `Card(union(Vars(s),Vars(t)))`. Assim,

temos o seguinte objetivo inicial de prova, sobre o qual invocamos o comando de prova `measure-induct+`:

- Sequente representado pelo nó **[raiz]**.

```
  |-------
{1}   FORALL (rho: Sub[variable, symbol, arity],
              s, t: term[variable, symbol, arity]):
        member(rho, U(s, t)) IMPLIES
           unification_algorithm(s, t) <= rho
```

Após aplicar a regra `measure-induct+`, o provador gera treze sub-objetivos de prova. O primeiro destes sub-objetivos consiste do ramo principal da árvore de prova, os outros sub-objetivos são resultado da checagem de tipos realizada pelo PVS. Na Figura 4.4.3, é possível visualizar a ramificação inicial da árvore de prova do Lema 4.4.2, onde consta o nó [raiz] e duas das ramificações geradas pela checagem de tipos. O ramo principal da prova é apresentado nas figuras 4.4.4 e 4.4.5.

Vamos apresentar a formalização do objetivo principal da prova. Apresentamos a árvore prova do ramo principal por partes. Na Figura 4.4.4 temos a primeira parte deste ramo. Começamos este ramo da árvore de prova com o seguinte sequente:

- Sequente representado pelo nó **[1]**.

```
{-1}  FORALL (y_1: term[variable, symbol, arity],
              y_2: term[variable, symbol, arity]):
        FORALL (rho: Sub[variable, symbol, arity]):
          Card(union(Vars(y_1), Vars(y_2))) <
           Card(union(Vars(x!1), Vars(x!2)))
           IMPLIES
           member(rho, U(y_1, y_2)) IMPLIES
            unification_algorithm(y_1, y_2) <= rho
  |-------
{1}   FORALL (rho: Sub[variable, symbol, arity]):
        member(rho, U(x!1, x!2)) IMPLIES
         unification_algorithm(x!1, x!2) <= rho
```

Note que este sequente é bastante semelhante ao apresentado na posição correspondente na árvore de prova do Lema 4.4.1. Então, temos a hipótese de indução apresentada na fórmula -1 e o nosso objetivo é provar o consequente 1. Assim, começamos aplicando a regra `skeep` para skolemizar a fórmula universalmente quantificada 1. Em seguida usamos

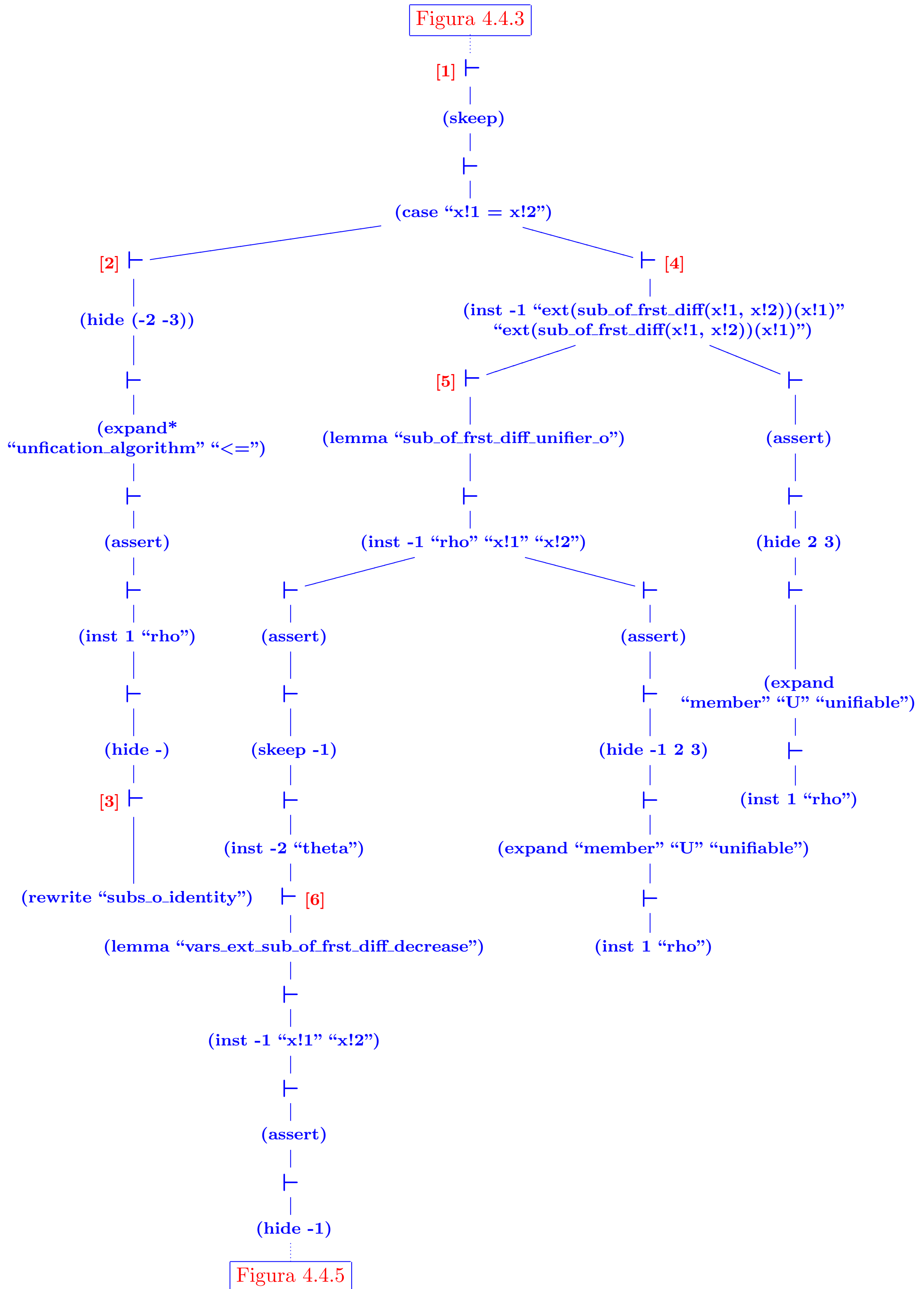


Figura 4.4.4: Início do ramo principal da árvore de prova do Lema 4.4.2. A parte final deste ramo da prova consta da Figura 4.4.5.

a regra `case`, para considerar o caso em que `x!1 = x!2` e o caso em que `x!1` $\neq$ `x!2`. Logo, neste ponto a prova se divide em dois sub-objetivos que apresentamos a seguir.

No primeiro caso, onde `x!1 = x!2`, temos o seguinte sub-objetivo de prova:

- Sequente representado pelo nó [2].

```
{-1}  x!1 = x!2
[-2]  FORALL (y_1: term[variable, symbol, arity],
              y_2: term[variable, symbol, arity]):
        FORALL (rho: Sub[variable, symbol, arity]):
          Card(union(Vars(y_1), Vars(y_2))) <
           Card(union(Vars(x!1), Vars(x!2)))
           IMPLIES
           member(rho, U(y_1, y_2)) IMPLIES
            unification_algorithm(y_1, y_2) <= rho
[-3]  member(rho, U(x!1, x!2))
  |-------
[1]   unification_algorithm(x!1, x!2) <= rho
```

Neste sequente começamos aplicando a regra `(hide -2 -3)`, que significa que estamos "escondendo" as fórmulas -2 e -3, onde -2 representa a hipótese de indução, mas não serão necessárias neste ramo da prova, pois como já comentamos anteriormente, no caso em que `x!1 = x!2` o construtor `unification_algorithm` retorna a substituição identidade. E note que a substituição identidade é mais geral que qualquer outra substituição. Assim, o que fazemos neste ramo da prova é expandir as definições de `unification_algorithm` e da pré-ordem `<=`, que usamos aqui para dizer que uma substituição é mais geral que outra. Feito isto e depois de realizar algumas simplificações, instanciamos a fórmula 1 com a substituição `rho` e obtemos o sequente:

- Sequente representado pelo nó [3]

```
  |-------
[1]   rho = comp(rho, identity)
```

Neste sequente apenas temos que nos dar conta de que a composição de qualquer substituição α com a substutuição identidade é igual a α. Este fato está formalizado no lema `subs_o_identity` da sub-teoria `substitution`. Assim, utilizamos a regra `rewrite` que procura uma instanciação para o lema invocado. Com isto completamos este ramo da prova.

Passemos ao caso em que `x!1` $\neq$ `x!2`, onde começamos com o objetivo de prova, dado pelo sequente:

- Sequente representado pelo nó [4]

```
[-1]  FORALL (y_1: term[variable, symbol, arity],
              y_2: term[variable, symbol, arity]):
        FORALL (rho: Sub[variable, symbol, arity]):
          Card(union(Vars(y_1), Vars(y_2))) <
           Card(union(Vars(x!1), Vars(x!2)))
           IMPLIES
           member(rho, U(y_1, y_2)) IMPLIES
            unification_algorithm(y_1, y_2) <= rho
[-2]  member(rho, U(x!1, x!2))
  |-------
{1}   x!1 = x!2
[2]   unification_algorithm(x!1, x!2) <= rho
```

Neste primeiro sequente deste ramo da prova, temos a hipótese que diferencia este caso do anterior na fórmula 1, que por estar no consequente representa a negação de `x!1 = x!2`. Além disso, temos a hipótese de indução na fórmula -1. Da mesma forma que fizemos no Lema 4.4.1, vamos começar instanciando -1 com os termos dados por `ext(sub_of_frst_diff(x!1, x!2))(x!1)` e `ext(sub_of_frst_diff(x!1, x!2))(x!2)`.

Feita esta instanciação, temos o seguinte sequente:

- Sequente representado pelo nó [5]

```
{-1}  FORALL (rho: Sub[variable, symbol, arity]):
        Card(union(Vars(ext(sub_of_frst_diff(x!1, x!2))(x!1)),
                   Vars(ext(sub_of_frst_diff(x!1, x!2))(x!2))))
         < Card(union(Vars(x!1), Vars(x!2)))
         IMPLIES
         member(rho,
                U(ext(sub_of_frst_diff(x!1, x!2))(x!1),
                  ext(sub_of_frst_diff(x!1, x!2))(x!2)))
          IMPLIES
          unification_algorithm(ext(sub_of_frst_diff(x!1, x!2))(x!1),
                                ext(sub_of_frst_diff(x!1, x!2))(x!2))
           <= rho
[-2]  member(rho, U(x!1, x!2))
  |-------
[1]   x!1 = x!2
[2]   unification_algorithm(x!1, x!2) <= rho
```

Note que depois de instanciar a hipótese de indução ainda temos uma fórmula com

uma variável quantificada universalmente, esta variável é do tipo `Sub`, isto é, uma substituição. Então precisamos instanciar esta fórmula com uma substituição específica. Vemos no corpo da fórmula que esta substituição deve ser um unificador dos termos `ext(sub_of_frst_diff(x!1, x!2))(x!1)` e `ext(sub_of_frst_diff(x!1, x!2))(x!2)`. Para obter esta substituição, ao invés de usar o lema `ext_sub_of_frst_diff_unifiable`, que estabelece que os termos são unificáveis, o que por skolemização nos daria uma substituição, escolhemos usar um o lema `sub_of_frst_diff_unifier_o`, onde afirmamos que para qualquer unificador dos termos s e t existe uma substituição θ, que composta com a substituição que resolve a primeira diferença dá este unificador. Usamos esta θ para instanciar a fórmula -1, e depois mostramos que θ é um unificador dos termos dados por `ext(sub_of_frst_diff(x!1,x!2))(x!1)` e `ext(sub_of_frst_diff(x!1, x!2))(x!2)`. Assim, após alguns passos de simplificação temos o sequente:

- Sequente representado pelo nó [6]

```
[-1]  rho = comp(theta, sub_of_frst_diff(x!1, x!2))
{-2}  Card(union(Vars(ext(sub_of_frst_diff(x!1, x!2))(x!1)),
                 Vars(ext(sub_of_frst_diff(x!1, x!2))(x!2))))
       < Card(union(Vars(x!1), Vars(x!2)))
       IMPLIES
       member(theta,
              U(ext(sub_of_frst_diff(x!1, x!2))(x!1),
                ext(sub_of_frst_diff(x!1, x!2))(x!2)))
        IMPLIES
        unification_algorithm(ext(sub_of_frst_diff(x!1, x!2))(x!1),
                              ext(sub_of_frst_diff(x!1, x!2))(x!2))
         <= theta
[-3]  member(rho, U(x!1, x!2))
  |-------
[1]   x!1 = x!2
[2]   unification_algorithm(x!1, x!2) <= rho
```

Observando a fórmula -2, que é resultado da hipótese de indução, temos que se

```
Card(union( Vars(ext(sub_of_frst_diff(x!1, x!2))x!1),
            Vars(ext(sub_of_frst_diff(x!1, x!2))x!2) ))
< Card(union( Vars(x!1), Vars(x!2) ))
```
(4.4.1)

e se a substituição θ, obtida através do lema usado anteriormente, é um unificador dos termos `ext(sub_of_frst_diff(x!1, x!2))(x!1)` e `ext(sub_of_frst_diff(x!1, x!2))(x!2)`, então a substituição dada por

```
unification_algorithm( ext(sub_of_frst_diff(x!1, x!2)),
                       ext(sub_of_frst_diff(x!1, x!2)) ),
```

é mais geral que θ.

Assim, novamente vamos usar o lema `vars_ext_sub_of_frst_diff_decrease`, que estabelece o resultado 4.4.1. Em seguida, após alguns passos de prova, realizamos uma simplificação proposicional com o uso da regra `prop`, o que faz com que tenhamos dois ramos na árvore de prova a partir deste ponto. A segunda parte da árvore de prova, que contém estes ramos está representada na Figura 4.4.5. No primeiro sequente do principal destes ramos temos:

- Sequente representado pelo nó [7]

```
{-1}  unification_algorithm(ext(sub_of_frst_diff(x!1, x!2))(x!1),
                            ext(sub_of_frst_diff(x!1, x!2))(x!2))
        <= theta
[-2]  rho = comp(theta, sub_of_frst_diff(x!1, x!2))
[-3]  member(rho, U(x!1, x!2))
  |-------
[1]   x!1 = x!2
[2]   unification_algorithm(x!1, x!2) <= rho
```

Neste sequente o nosso objetivo é provar a assertiva da fórmula 2. Para isto, expandimos `unification_algorithm` na fórmula 2, em seguida expandimos `<=`, que nos gera fórmulas quantificadas existencialmente. Neste ponto temos o sequente:

- Sequente representado pelo nó [8]

```
[-1]  unification_algorithm(ext(sig1)(x!1), ext(sig1)(x!2)) = sig2
[-2]  sub_of_frst_diff(x!1, x!2) = sig1
{-3}  EXISTS tau: theta = comp(tau, sig2)
[-4]  rho = comp(theta, sig1)
[-5]  member(rho, U(x!1, x!2))
  |-------
[1]   x!1 = x!2
{2}   EXISTS tau: rho = comp(tau, comp(sig2, sig1))
```

Depois de skolemizar a variável quantificada em -3, instanciamos 2 com esta mesma variável. Assim, após alguns passos de prova e algumas simplificações, temos o objetivo:

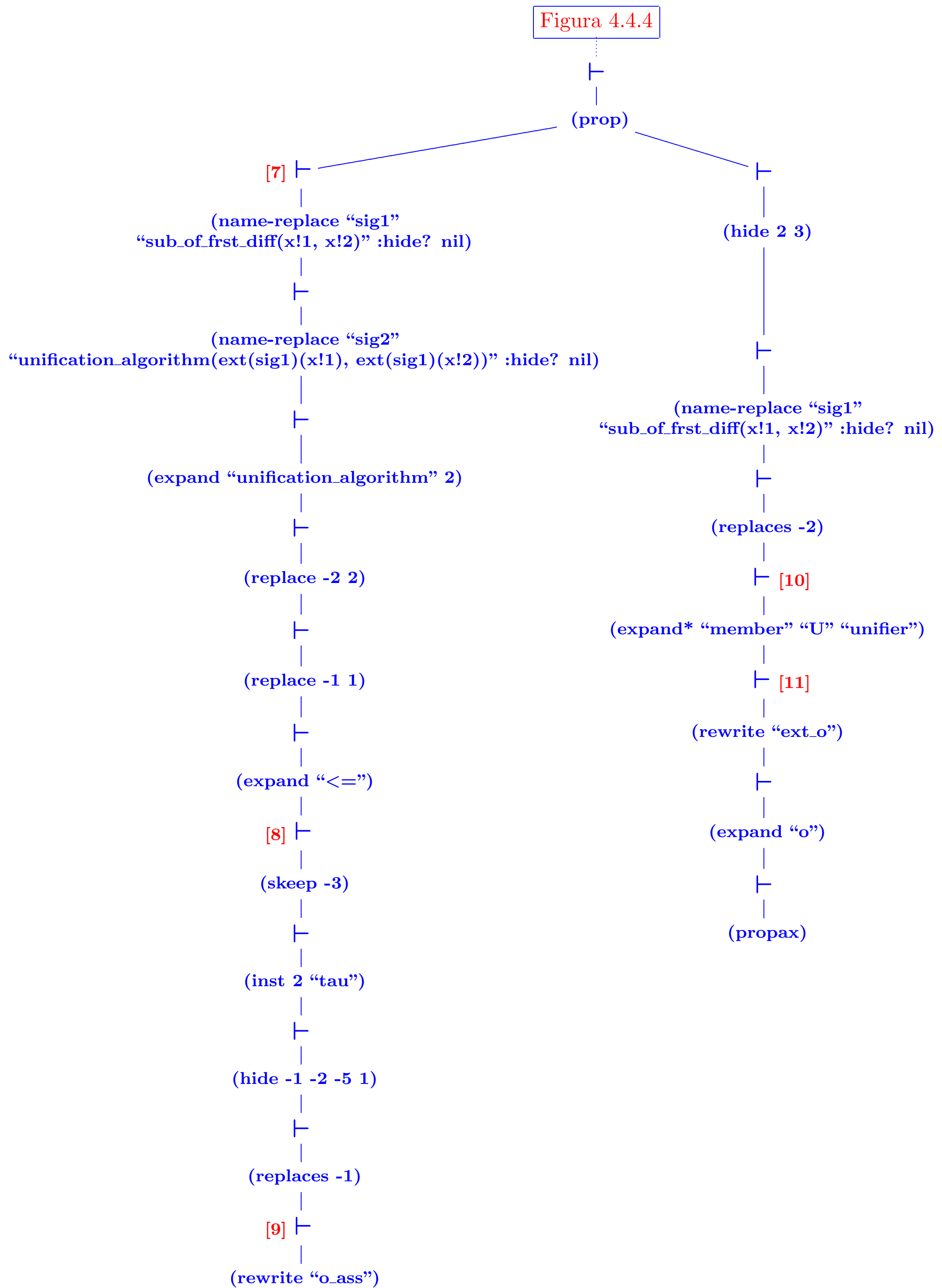


Figura 4.4.5: Parte final do ramo principal da árvore de prova do Lema 4.4.2. A parte inicial deste ramo da prova consta da Figura 4.4.4.

- Sequente representado pelo nó [9]

```
{-1}  rho = comp(comp(tau, sig2), sig1)
  |-------
{1}   rho = comp(tau, comp(sig2, sig1))
```

Neste sequente, temos que:

`sig1 = sub_of_frst_diff(x!1, x!2)` e

`sig2 = unification_algorithm(ext(sig1)(x!1), ext(sig1)(x!2))`.

Note que a fórmula -1 é igual à fórmula 1, apenas temos que nos dar conta de que a composição de substituições é associativa, isto é, dadas as substituições τ, σ_1 e σ_2, vale que $((\tau \circ \sigma_1) \circ \sigma_2) = (\tau \circ (\sigma_1 \circ \sigma_2))$. Mas este resultado esta formalizado no lema `o_ass` da *sub-teoria* `substitution`.

No ramo seguinte da prova, temos o seguinte objetivo:

- Sequente representado pelo nó [10]

```
{-1}  sub_of_frst_diff(x!1, x!2) = sig1
{-2}  member(comp(theta, sig1), U(x!1, x!2))
  |-------
{1}   member(theta, U(ext(sig1)(x!1), ext(sig1)(x!2)))
```

Neste sequente devemos verificar que a substituição obtida pela aplicação do lema `sub_of_frst_diff_unifier_o`, é de fato um unificador dos termos `ext(sig1)(x!1)` e `ext(sig1)(x!2)`. Para isto, apenas expandimos as definições de `meber`, `U` e `unifier` que aparece devido as expansões anteriores. Feito isto, temos o objetivo:

- Sequente representado pelo nó [11]

```
[-1]  sub_of_frst_diff(x!1, x!2) = sig1
{-2}  ext(comp(theta, sig1))(x!1) = ext(comp(theta, sig1))(x!2)
  |-------
{1}   ext(theta)(ext(sig1)(x!1)) = ext(theta)(ext(sig1)(x!2))
```

Onde temos que verificar que dadas duas substituições θ e σ quaisquer, vale que $\widehat{\theta \circ \sigma} = \hat{\theta} \circ \hat{\sigma}$. Mas este resultado está formalizado no lema `ext_o` da *sub-teoria* `substitution`. Assim, reescrevemos com este lema e completamos este ramo da prova.

Com isto, completamos o ramo principal da prova. Ressaltamos que nos ramos da prova descritos acima, omitimos alguns sub-objetivos, que aparecem como resultado da checagem de tipos realizada pelo PVS, usamos a estratégia de aplicar um outro lema da teoria, nestes casos temos que provar asserções acerca do tipo dos termos.

Em todos os ramos seguintes da prova, temos que provar asserções acerca de tipos. Em alguns destes objetivos temos que provar que a união dos conjuntos de variáveis dos termos é finita. A demonstração destes ramos, assim como fizemos na demonstração do Lema 4.4.1, segue pela aplicação de dois outros lemas, um da teoria `finite_sets` do prelúdio do PVS, denominado `finite_union` e um da teoria `subterm` denominado `vars_of_term_finite`. Como já descrevemos este ramo de prova na apresentação da formalização do Lema 4.4.1, não repetiremos aqui. Nos outros ramos da prova, temos apenas que mostrar que os termos são unificáveis. Assim, temos sempre um sequente da forma:

- Sequente representado pelo nó [12]

```
[-1]  member(rho!1, U(s!1, t!1))
  |-------
[1]   unifiable(s!1, t!1)
```

Note que na fórmula -1 temos que a substituição `rho!1` faz parte do conjunto de unificadores dos termos `s!1` e `t!1`. Assim, expandimos as definições de `member`, `U` e `unifiable`, em seguida instanciamos a fórmula 1, que passa a ser quantificada existencialmente, com a substituição `rho!1` o que completa este ramo da prova.

Com isto completamos todos os ramos da prova do lema. ■

4.5 Terminação e Formalização de Lemas Sobre o Construtor `sub_of_frst_diff`

Nesta seção apresentamos a formalização dos lemas sobre o construtor `sub_of_frst_diff`, cuja especificação consta na Tabela 4.2.6. A formalização destes lemas surge da necessidade de verificar que o construtor `unification_algorithm` está totalmente definido em relação aos tipos dos termos. Além disso, com estes lemas, verificamos propriedades essenciais do algoritmo de unificação, uma delas é a de terminação. A terminação do algoritmo

proposto nesta especificação fica garantida pelo lema que apresentamos primeiro nesta seção, o Lema 4.5.1. Nos lemas 4.5.2 e 4.5.3, apresentados nas seções 4.5.2 e 4.5.3, formalizamos o fato de que resolver diferenças em termos unificáveis leva a termos ainda unificáveis. Estes foram os lemas *essenciais* no passo indutivo da demosntração de cada um dos lemas apresentados na Seção 4.4. Os lemas desta seção correspodem aos lemas sobre generalidade local e terminação do algoritmo de unificação apresentados na Seção 2.2.3.

4.5.1 Lema `vars_ext_sub_of_frst_diff_decrease`

Como já mencionamos anteriormente, com este lema temos garantida a terminação do algoritmo de unificação, isto se deve ao fato de que a cada passo do algoritmo retiramos um elemento de um conjunto finito, o que portanto é um processo finito. A especificação deste lema está apresentada na Tabela 4.2.6. Na Seção 2.2, demonstramos o Lema 2.2.15, onde provamos a terminação do algoritmo de unificação e que corresponde a este cuja formalização apresentamos aqui. A árvore de prova deste lema é apresentada nas figuras 4.5.1, 4.5.2 e 4.5.3.

Lema 4.5.1: *Sejam s e t dois termos unificáveis e $\sigma =$* `sub_of_frst_diff`(s,t)*, então*

$$Card(\mathcal{V}ars(\hat{\sigma}(s)) \cup \mathcal{V}ars(\hat{\sigma}(t))) < Card(\mathcal{V}ars(s) \cup \mathcal{V}ars(t)).$$

Demonstração: Na formalização deste lema temos o seguinte objetivo de prova, ao iniciarmos a demonstração:

- Sequente representado pelo nó [**raiz**]

```
  |-------
{1}   FORALL (s: term, t: term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
          Card(union(Vars(ext(sig)(s)), Vars(ext(sig)(t)))) <
           Card(union(Vars(s), Vars(t)))
```

Começamos skolemizando as variáveis ligadas do consequente com a regra `skosimp`, que ao contrário da regra `skeep`, skolemiza introduzindo novos nomes para as constantes, o que

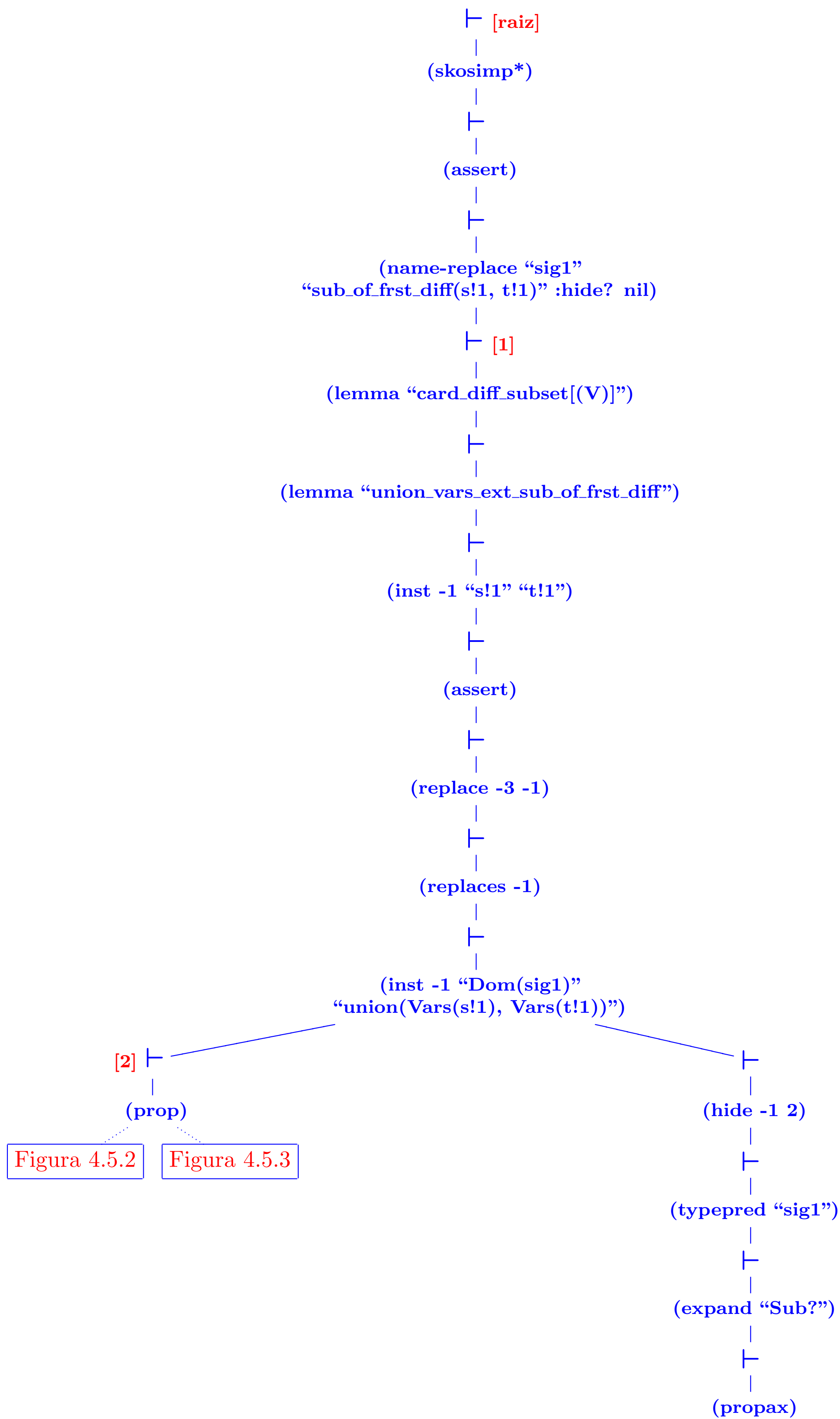


Figura 4.5.1: Início da árvore de prova do Lema 4.5.1. O restante da árvore consta das figuras 4.5.2 e 4.5.3.

evita conflitos entre os nomes. Após poucos passos de prova onde fazemos simplificações, temos:

- Sequente representado pelo nó [1]

```
{-1}  sub_of_frst_diff(s!1, t!1) = sig1
  |-------
{1}   Card(union(Vars(ext(sig1)(s!1)), Vars(ext(sig1)(t!1)))) <
       Card(union(Vars(s!1), Vars(t!1)))
```

Note que estamos denotando a substituição que resolve a primeira diferença por `sig1`. Para chegar a uma contradição a partir de 1, utilizamos a seguinte estratégia: mostramos em um outro lema, denominado `union_vars_ext_sub_of_frst_diff`, que

$$\mathcal{V}ars(\hat{\sigma}(s)) \cup \mathcal{V}ars(\hat{\sigma}(t)) = \mathcal{V}ars(s) \cup \mathcal{V}ars(t) \setminus \mathcal{D}om(\sigma),$$

e utilizamos o lema `card_diff_subset` da *teoria* `finite_sets` da biblioteca do prelúdio do PVS, onde se afirma que dados dois conjuntos A e B, vale que

$$A \subset B \Rightarrow Card(B \setminus A) = Card(B) - Card(A).$$

Assim, utilizando estes dois lemas, e após alguns passo de prova, temos:

- Sequente representado pelo nó [2]

```
{-1}  subset?(Dom(sig1), union(Vars(s!1), Vars(t!1))) IMPLIES
       card(difference(union(Vars(s!1), Vars(t!1)), Dom(sig1))) =
        card(union(Vars(s!1), Vars(t!1))) - card(Dom(sig1))
[-2]  sub_of_frst_diff(s!1, t!1) = sig1
  |-------
[1]   Card(difference(union(Vars(s!1), Vars(t!1)), Dom(sig1))) <
       Card(union(Vars(s!1), Vars(t!1)))
```

A este sequente aplicamos a regra `prop`, que realiza uma simplificação proposicional e divide a prova neste ponto em dois ramos. O primeiro destes ramos começa com o seguinte objetivo:

- Sequente representado pelo nó [3]

```
{-1}  card(difference(union(Vars(s!1), Vars(t!1)), Dom(sig1))) =
       card(union(Vars(s!1), Vars(t!1))) - card(Dom(sig1))
[-2]  sub_of_frst_diff(s!1, t!1) = sig1
  |-------
[1]   Card(difference(union(Vars(s!1), Vars(t!1)), Dom(sig1))) <
       Card(union(Vars(s!1), Vars(t!1)))
```

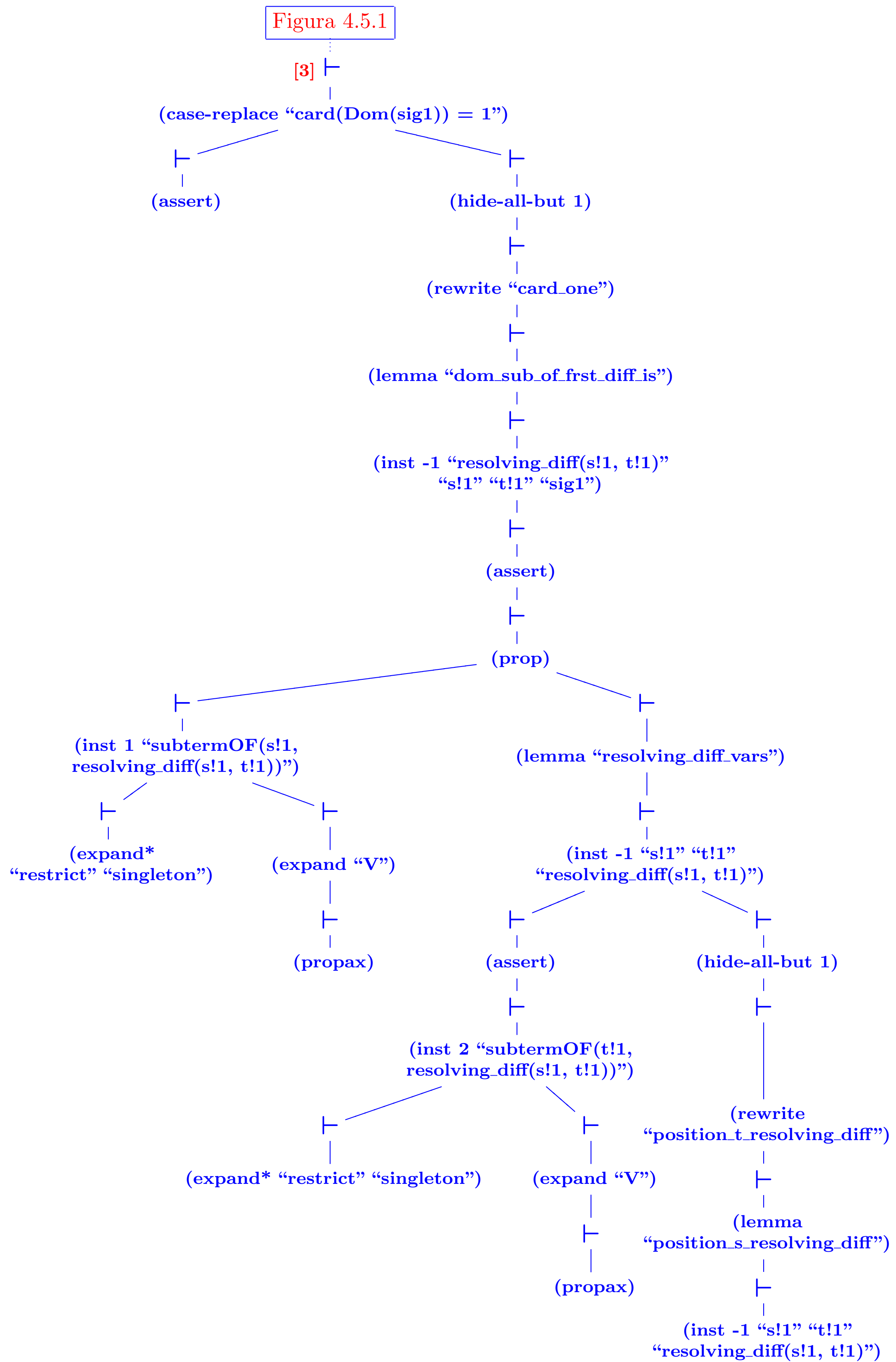


Figura 4.5.2: Parte da árvore de prova do Lema 4.5.1.

Neste ponto consideramos o caso em que `card(Dom(sig1)) = 1`. O que nos leva a outros dois ramos de prova. O primeiro completamos com um `assert`, pois neste caso -1 é uma contradição de 1. E com isto completamos o ramo principal da prova. No segundo caso, temos que provar que de fato `card(Dom(sig1)) = 1`, mas já temos formalizado no lema `dom_sub_of_frst_diff_is`, o fato de que o domínio da substituição que resolve a primeira diferença é unitário. Assim, partindo deste lema, completamos este ramo da prova, e depois de verificar outros ramos gerados pelo *typechecker*, chegamos ao seguinte:

- Sequente representado pelo nó [4]

```
[-1]  sub_of_frst_diff(s!1, t!1) = sig1
  |-------
{1}   subset?(Dom(sig1), union(Vars(s!1), Vars(t!1)))
[2]   Card(difference(union(Vars(s!1), Vars(t!1)), Dom(sig1))) <
       Card(union(Vars(s!1), Vars(t!1)))
```

Este sequente representa o segundo objetivo gerado pela aplicação da regra `prop`. Neste queremos concluir que se $\sigma =$ `sub_of_frst_diff`(s, t), então

$$Dom(\sigma) \ \subset \ \mathcal{V}ars(s) \cup \mathcal{V}ars(t). \qquad (4.5.1)$$

Este objetivo esta representado pela fórmula 1 do sequente acima. Para concluir este ramo da prova utilizamos novamente o lema `dom_sub_of_frst_diff_is`, onde mostramos também que se p é a posição da primeira diferença entre os termos s e t, então temos apenas duas possibilidades exclusivas para o domininío de σ, ou $Dom(\sigma) = \{s|_p\}$ ou $Dom(\sigma) = \{t|_p\}$. Em ambos os casos temos que 4.5.1 é satisfeita, pois

$$\begin{aligned} Dom(\sigma) = \{s|_p\} \ &\Rightarrow \ Dom(\sigma) \ \subset \ \mathcal{V}ars(s) \\ &\Rightarrow \ Dom(\sigma) \ \subset \ \mathcal{V}ars(s) \cup \mathcal{V}ars(t) \end{aligned}$$

ou

$$\begin{aligned} Dom(\sigma) = \{t|_p\} \ &\Rightarrow \ Dom(\sigma) \ \subset \ \mathcal{V}ars(t) \\ &\Rightarrow \ Dom(\sigma) \ \subset \ \mathcal{V}ars(s) \cup \mathcal{V}ars(t) \end{aligned}$$

Assim, antes de aplicarmos o lema mencionado, realizamos alguns passos de prova, onde expandimos as definições de `subset?`, `member` e `union`, aplicamos simplificações proposicionais e decompomos uma igualdade chegando ao seguinte objetivo de prova:

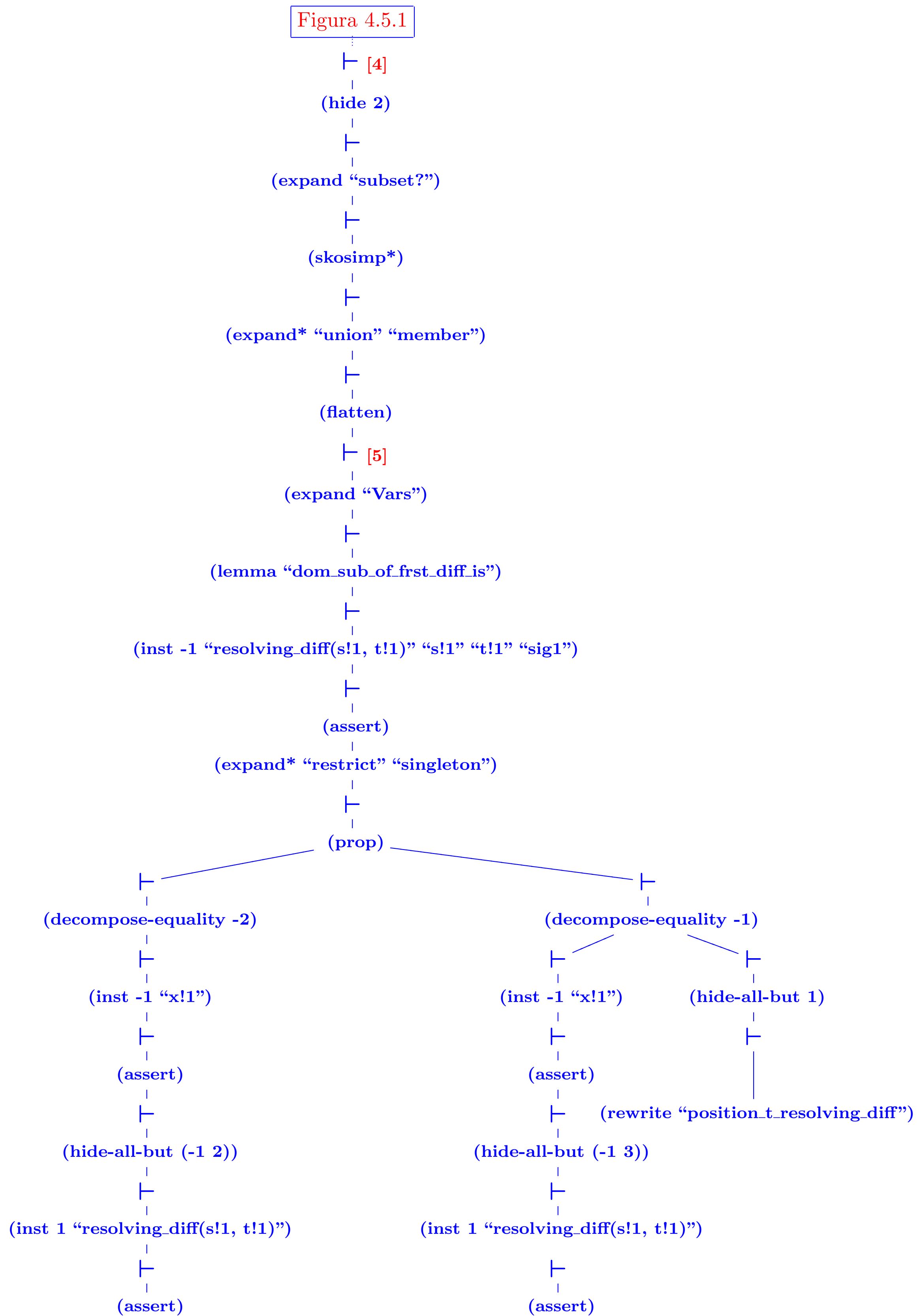


Figura 4.5.3: Parte da árvore de prova do Lema 4.5.1.

- Sequente representado pelo nó [5]

```
[-1]  Dom(sig1)(x!1)
[-2]  sub_of_frst_diff(s!1, t!1) = sig1
  |-------
{1}   Vars(s!1)(x!1)
{2}   Vars(t!1)(x!1)
```

A variável `x!1` surge quando expandimos a definição de `subset?` e skolemizamos a variável da fórmula quantificada universalmente que obtemos no consequente. Assim, o que queremos neste objetivo é mostrar que se `x!1` é uma variável do domínio de `sig1` (fórmula -1), então ou `x!1` é uma variável de `Vars(s!1)` (fórmula 1) ou `x!1` é uma variável de `Vars(t!1)` (fórmula 2). Neste ponto utilizamos o lema `dom_sub_of_frst_diff_is`, e após alguns passos de prova, onde realizamos simpificações completamos este ramo da prova, que também possui sub-ramos onde provamos alguns TCC's gerados pelo *typechecker*. ■

4.5.2 Lema `sub_of_frst_diff_unifier_o`

Neste lema, provamos que sempre que uma substituição pertence ao conjunto de unificadores de dois termos unificáveis, então existe uma outra substituição que composta com aquela computada em um passo de execução do algoritmo de unificação, resulta na primeira. Este lema corresponde ao Lema 2.2.14, sobre generalidade local do algoritmo de unificação. Na Tabela 4.2.6 temos a especificação do lema seguinte e nas figuras 4.5.4, 4.5.5 e 4.5.6 a sua árvore de prova.

Lema 4.5.2: *Sejam s e t termos unificáveis e diferentes e σ a substituição que resolve a primeira diferença entre os termos s e t, isto é, $\sigma =$ `sub_of_frst_diff`(s, t). Então, para todo unificador $\theta \in \mathcal{U}(s, t)$, existe uma substituição δ tal que $\theta = \delta \circ \sigma$.*

Demonstração: Na demonstração deste lema, basicamente vamos mostrar que as substituições θ e $\delta \circ \sigma$ possuem o mesmo domínio e a mesma imagem. Temos o seguinte objetivo inicial de prova:

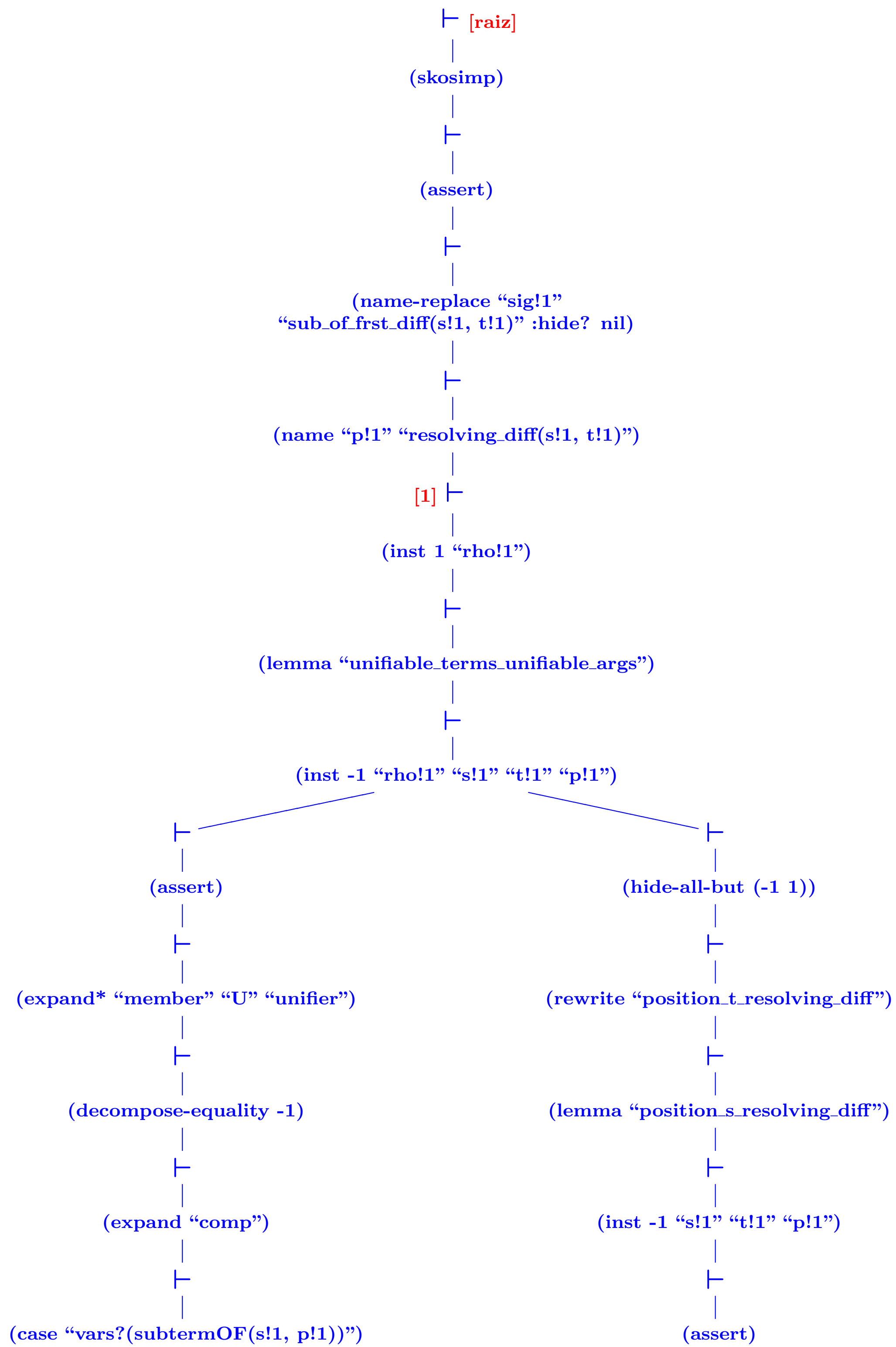


Figura 4.5.4: Ramificação inicial da árvore de prova do Lema 4.5.2. A parte final da árvore de prova consta das figuras 4.5.5 e 4.5.6.

- Sequente representado pelo nó [raiz]

```
  |-------
{1}   FORALL (rho: Sub[variable, symbol, arity], s: term,
              t: term | unifiable(s, t) & s /= t):
        member(rho, U(s, t)) IMPLIES
         LET sig = sub_of_frst_diff(s, t) IN
           EXISTS theta: rho = comp(theta, sig)
```

Começamos skolemizando as variáveis ligadas da fórmula 1. Após alguns passos temos o seguinte sequente, onde é possível visualizar de forma mais clara o nosso objetivo de prova.

- Sequente representado pelo nó [1]

```
{-1}  resolving_diff(s!1, t!1) = p!1
[-2]  sub_of_frst_diff(s!1, t!1) = sig!1
[-3]  member(rho!1, U(s!1, t!1))
  |-------
[1]   EXISTS theta: rho!1 = comp(theta, sig!1)
```

Temos uma substituição `rho!1`, que pela fórmula -3, é um unificador de `s!1` e `t!1`. Queremos chegar ao resultado da fórmula 1, isto é, existe uma substituição `theta` que composta com `sig!1` resulta em `rho!1` e, como vemos em -2, `sig!1` é a substituição que resolve a primeira diferença entre `s!1` e `t!1`. A estratégia é instanciar a fórmula 1 com a própria `rho!1`. Mas como `rho!1` unifica `s!1` e `t!1`, temos em mente que

```
member(rho!1, U(s!1, t!1)) IMPLIES
       member(rho!1, U(subtermOF(s!1, q!1), subtermOF(t!1, q!1))),
```

onde `q!1` é uma posição qualquer dos termos `s!1` e `t!1`. Assim, o que faremos é observar o caso particular em que `q!1 = p!1`, onde vemos na fórmula -1 que `p!1` é a posição onde ocorre a primeira diferença entre os termos `s!1` e `t!1`. Como já sabemos que na posição `p!1` ou o subtermo de `s!1` é uma variável ou o subtermo de `t!1` é uma variável, uma boa estratégia é analizar cada um destes casos. Assim prosseguimos e temos então ramificações na árvore de prova. Após alguns passos de prova, começamos pelo caso em que `subtermOF(s!1,p!1)` é uma variável.

- Sequente representado pelo nó [2]

```
{-1}  vars?(subtermOF(s!1, p!1))
[-2]  ext(rho!1)(subtermOF(s!1, p!1)) = ext(rho!1)(subtermOF(t!1, p!1))
[-3]  resolving_diff(s!1, t!1) = p!1
[-4]  sub_of_frst_diff(s!1, t!1) = sig!1
[-5]  ext(rho!1)(s!1) = ext(rho!1)(t!1)
  |-------
[1]   rho!1(x!1) = ext(rho!1)(sig!1(x!1))
```

Observe que temos uma nova variável `x!1` na fórmula 1, que aparece quando decompomos a igualdade desta fórmula, isto é, temos que se `rho!1 = comp(theta, sig!1)`, então `rho!1(x!1) = ext(rho!1)(sig!1(x!1))`, para qualquer variável `x!1`. Então neste ponto da prova, novamente vamos considerar casos. Primeiro o caso em que `x!1 = subtermOF(s!1, p!1)`. Neste caso, observe que pela definição de `sub_of_frst_diff` temos que:

- Sequente representado pelo nó [3]

```
{-1}  x!1 = subtermOF(s!1, p!1)
[-2]  vars?(subtermOF(s!1, p!1))
[-3]  ext(rho!1)(subtermOF(s!1, p!1)) = ext(rho!1)(subtermOF(t!1, p!1))
[-4]  resolving_diff(s!1, t!1) = p!1
[-5]  sub_of_frst_diff(s!1, t!1) = sig!1
[-6]  ext(rho!1)(s!1) = ext(rho!1)(t!1)
  |-------
[1]   rho!1(x!1) = ext(rho!1)(sig!1(x!1))
```

Deste sequente, tiramos as seguintes conclusões:

Por definição de `sig!1`, temos que

```
ext(sig!1)(subtermOF(s!1, p!1)) = subtermOF(t!1, p!1).
```

Além disso, estamos no caso em que `x!1 = subtermOF(s!1, p!1)` e sendo `rho!1` um unificador de `s` e `t`, então

```
ext(rho!1)(subtermOF(s!1, p!1)) = ext(rho!1)(subtermOF(t!1,p!1)).
```

Logo, concluímos que

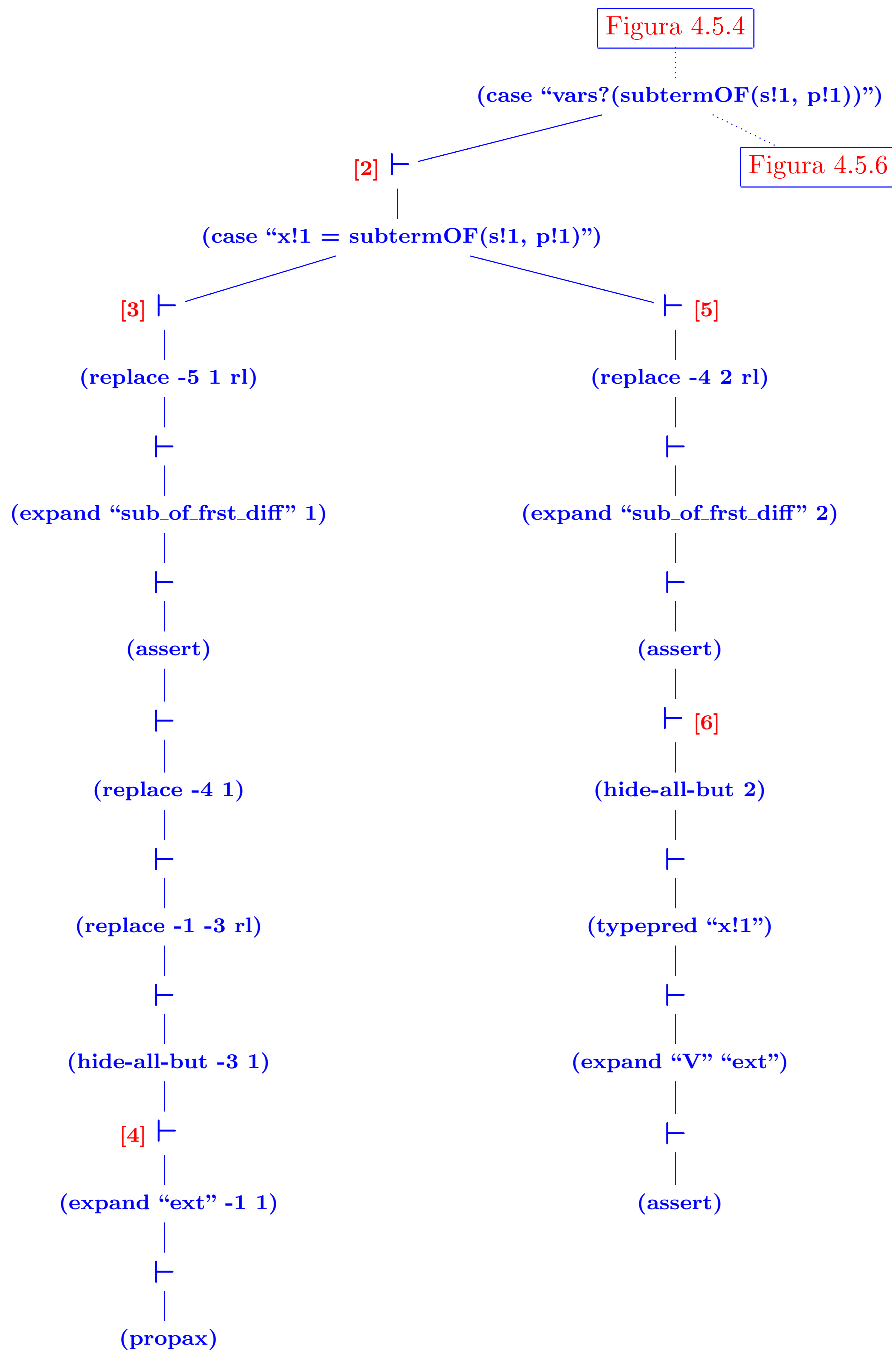


Figura 4.5.5: Parte da árvore de prova do Lema 4.5.2. Representa o caso em que o termo `subtermOF(s!1, p!1)` é uma variável.

```
    ext(rho!1)(x!1) = ext(rho!1)(subtermOF(t!1, p!1))
⇒   ext(rho!1)(x!1) = ext(rho!1)(ext(sig!1)(subtermOF(s!1,p!1)))
⇒   ext(rho!1)(x!1) = ext(rho!1)(ext(sig!1)(x!1)),
```

onde a última expressão é justamente a fórmula 1 do sequente anterior. Assim, realizando alguns passos de prova, onde reproduzimos o raciocínio descrito acima, obtemos o

sequente:

- Sequente representado pelo nó [4]

```
[-1]   ext(rho!1)(x!1) = ext(rho!1)(subtermOF(t!1, p!1))
  |-------
[1]    rho!1(x!1) = ext(rho!1)(subtermOF(t!1, p!1))
```

Note que temos uma contradição pois na fórmula 1 temos a negação de -1. Para concluir este ramo da prova basta expandir a definição de `ext`, pois como `x!1` é uma variável, então `ext(rho!1)(x!1) = rho!1(x!1)`. Com isto completamos este ramo da prova.

No próximo ramo estamos no caso em que `x!1` $\neq$ `subtermOF(s!1, p!1)`. O sequente inicial é dado por:

- Sequente representado pelo nó [5]

```
[-1]   vars?(subtermOF(s!1, p!1))
[-2]   ext(rho!1)(subtermOF(s!1, p!1)) = ext(rho!1)(subtermOF(t!1, p!1))
[-3]   resolving_diff(s!1, t!1) = p!1
[-4]   sub_of_frst_diff(s!1, t!1) = sig!1
[-5]   ext(rho!1)(s!1) = ext(rho!1)(t!1)
  |-------
{1}    x!1 = subtermOF(s!1, p!1)
[2]    rho!1(x!1) = ext(rho!1)(sig!1(x!1))
```

Neste caso a demonstração é mais direta, pois como `x!1` é uma variável que não pertence ao domínio de `sig!1`, então `sig!1(x!1) = x!1`. Assim, basta expandir a definição de `sig!1` em 2 e teremos diretamente que `rho!1(x!1) = ext(rho!1)(x!1)`. Portanto, após alguns passos de prova, temos o sequente:

- Sequente representado pelo nó [6]

```
[-1]   vars?(subtermOF(s!1, p!1))
[-2]   ext(rho!1)(subtermOF(s!1, p!1)) = ext(rho!1)(subtermOF(t!1, p!1))
[-3]   resolving_diff(s!1, t!1) = p!1
[-4]   sub_of_frst_diff(s!1, t!1) = sig!1
[-5]   ext(rho!1)(s!1) = ext(rho!1)(t!1)
  |-------
[1]    x!1 = subtermOF(s!1, p!1)
{2}    rho!1(x!1) = ext(rho!1)(x!1)
```

Note que neste sequente a fórmula principal é o consequente 2. Assim, como já mencionamos, expandimos `ext` e com isto completamos este ramo da prova.

No ramo seguinte consideramos o caso em que o termo `subtermOF(t!1, p!1)` é uma variável. Este ramo é inteiramente análogo ao que descrevemos acima, e não o exporemos aqui.

Passando ao ramo seguinte, temos o seguinte sequente:

- Sequente representado pelo nó [7].

```
[-1]   ext(rho!1)(subtermOF(s!1, p!1)) = ext(rho!1)(subtermOF(t!1, p!1))
[-2]   resolving_diff(s!1, t!1) = p!1
[-3]   sub_of_frst_diff(s!1, t!1) = sig!1
[-4]   ext(rho!1)(s!1) = ext(rho!1)(t!1)
  |-------
{1}    vars?(subtermOF(t!1, p!1))
[2]    vars?(subtermOF(s!1, p!1))
[3]    rho!1(x!1) = ext(rho!1)(sig!1(x!1))
```

Observe que neste ramo estamos considerando o caso em que nenhum dos termos `subtermOF(t!1, p!1)` e `subtermOF(s!1, p!1)` é uma variável. Mas já sabemos que este caso não ocorre, e este fato está formalizado no lema `resolving_diff_vars` , cuja especificação apresentamos na Tabela 4.2.4. Assim, com este lema após feitas as devidas instanciações temos o sequente:

- Sequente representado pelo nó [8]

```
{-1}   p!1 = resolving_diff(s!1, t!1) IMPLIES
         vars?(subtermOF(s!1, p!1)) OR vars?(subtermOF(t!1, p!1))
[-2]   resolving_diff(s!1, t!1) = p!1
  |-------
[1]    vars?(subtermOF(t!1, p!1))
[2]    vars?(subtermOF(s!1, p!1))
```

Note que temos uma contradição, pois pela fórmula 1, `subtermOF(t!1, p!1)` não é uma variável, e pela fórmula 2, `subtermOF(s!1, p!1)` não é uma variável, mas pela fórmula -1, ou `subtermOF(s!1, p!1)` é uma variável ou `subtermOF(t!1, p!1)` é uma variável. Assim, com uma aplicação da regra `assert`, completamos este ramo da prova.

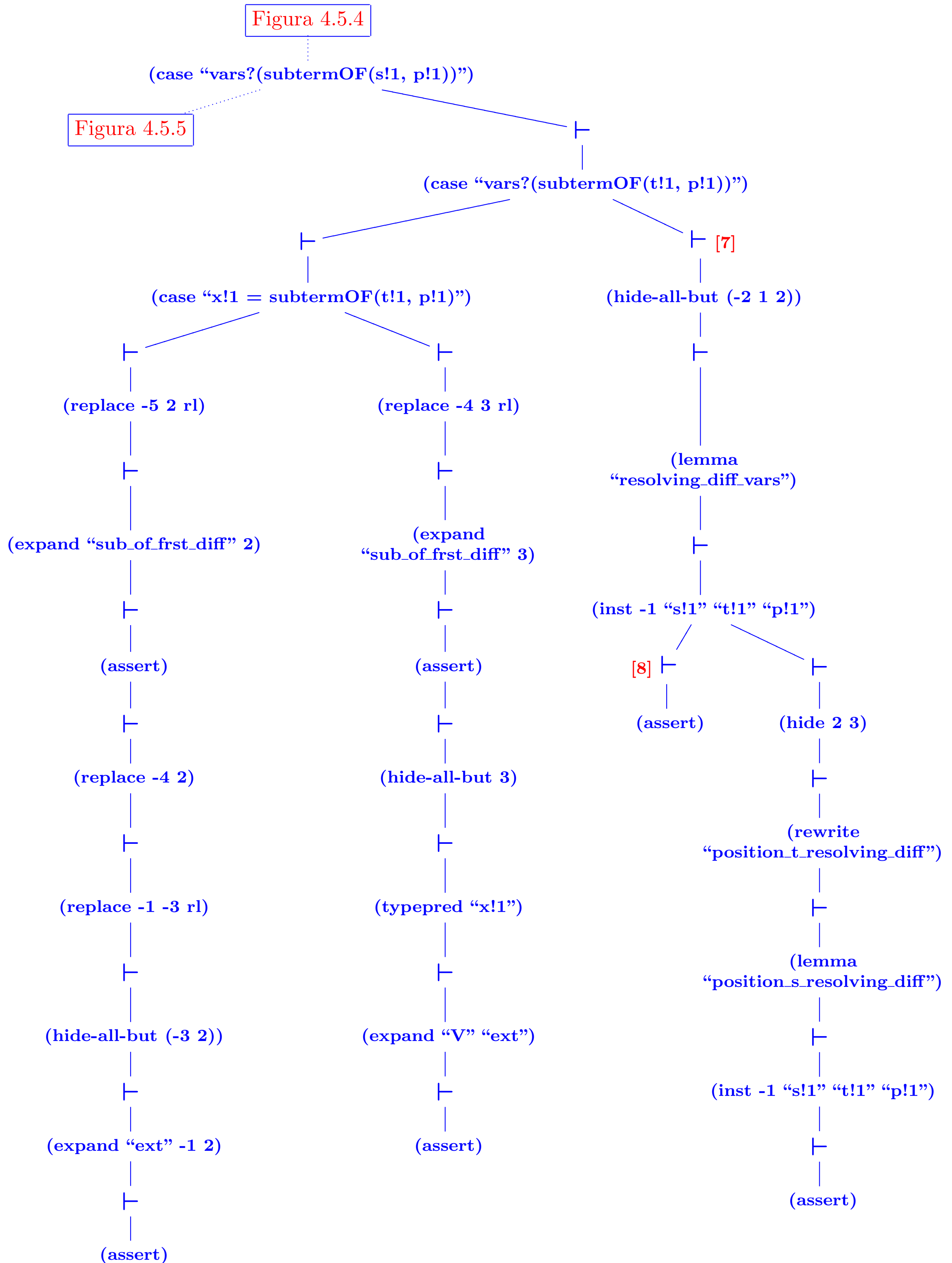


Figura 4.5.6: Parte da árvore de prova do Lema 4.5.2. Representa o caso em que o termo `subtermOF(s!1, p!1)` não é uma variável. A ramificação abaixo no nó [7], representa o caso em que nem `subtermOF(s!1, p!1)` e nem `subtermOF(t!1, p!1)` são variáveis.

Novamente apresentamos apenas o ramo principal da formalização do Lema 4.5.2, denominado `sub_of_frst_unifier_o`. Outros ramos gerados pela operação de checagem de tipos não serão comentados aqui. ■

4.5.3 Lema `ext_sub_of_frst_diff_unifiable`

Nesta seção apresentamos a formalização do lema `ext_sub_of_frst_diff_unifiable`, onde provamos que os termos obtidos via instanciação, de dois termos unificáveis e diferentes, pela substituição computada pelo construtor `sub_of_frst_diff` aplicado a estes dois termos, são ainda termos unificáveis.

O lema `sub_of_frst_diff_unifiable` foi necessário na formalização do Lema 4.4.1, denominado `unification_algorithm_gives_unifier`. Também foi necessário na demonstração de um importante TCC sobre o construtor `unification_algorithm`, onde deve-se verificar que o construtor preserva a tipagem dos termos, no sentido de que os novos termos gerados no processo de recursão devem ser ainda unificáveis.

O lema `sub_of_frst_diff_unifiable` corresponde a um dos resultados do Lema 2.2.14, demonstrado no Capítulo 2, onde vimos a sua importância na verificação da completude do algoritmo de unificação. A especificação deste lema consta da Tabela 4.2.6. A formalização deste lema é bastante simples, pois resulta de uma aplicação direta do Lema 4.5.2. Assim, apresentaremos a formalização deste lema sem expor a árvore de prova, que possui apenas um ramo.

Lema 4.5.3: *Sejam s e t termos unificáveis e diferentes e σ a substituição que resolve a primeira diferença entre os termos s e t, isto é, $\sigma =$* `sub_of_frst_diff`(s, t)*. Então, os termos $\hat{\sigma}(s)$ e $\hat{\sigma}(t)$ são ainda unificáveis.*

Demonstração: Começamos a demonstração deste lema com o seguinte objetivo de prova:

```
  |-------
{1}   FORALL (s: term, t: term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
          unifiable(ext(sig)(s), (ext(sig)(t)))
```

Como já mencionamos anteriormente, a formalização deste lema segue diretamente do Lema 4.5.2. De fato, o lema `ext_sub_of_frst_diff_unifiable` é um corolário do lema `sub_of_frst_diff_unifier_o`. Assim, após skolemizar as variáveis do sequente acima, utilizamos a regra `lemma`, a fim de trazer para as fórmulas do antecedente a hipótese do lema `sub_frst_diff_unifier_o`, e com isto obtemos o seguinte:

```
{-1}  FORALL (rho: Sub[variable, symbol, arity], s: term,
                t: term | unifiable(s, t) & s /= t):
        member(rho, U(s, t)) IMPLIES
         LET sig = sub_of_frst_diff(s, t) IN
           EXISTS theta: rho = comp(theta, sig)
[-2]  sub_of_frst_diff(s!1, t!1) = sig1
  |-------
[1]   unifiable(ext(sig1)(s!1), (ext(sig1)(t!1)))
```

Note que a fórmula -1 esta quantificada universalmente, e devemos instanciá-la adequadamente com uma substituição que seja um unificador dos termos `s!1 s!2`. Esta substituição sairá do fato de que estes dois termos são unificáveis. Assim, após alguns passos de prova, obtemos:

```
[-1]  unifier(sigma)(s!1, t!1)
{-2}  EXISTS theta: sigma = comp(theta, sig1)
[-3]  sub_of_frst_diff(s!1, t!1) = sig1
  |-------
[1]   s!1 = t!1
[2]   EXISTS sigma: unifier(sigma)(ext(sig1)(s!1), (ext(sig1)(t!1)))
```

Neste ponto da prova temos a fórmula -2, que é uma consequência direta do lema `sub_frst_diff_unifier_o`, onde temos que existe uma substituição que composta com `sig1` resulta no unificador `sigma`. Então skolemizamos esta fórmula e usamos a variável skolemizada para instanciar a fórmula 2. Após alguns passos de prova, temos:

```
{-1}  ext(comp(theta, sig1))(s!1) = ext(comp(theta, sig1))(t!1)
[-2]  sub_of_frst_diff(s!1, t!1) = sig1
  |-------
{1}   ext(theta)(ext(sig1)(s!1)) = ext(theta)((ext(sig1)(t!1)))
```

Assim, como feito no Lema 4.4.2, temos que verificar que dadas duas substituições θ e σ quaiquer, vale que $\widehat{\theta \circ \sigma} = \hat{\theta} \circ \hat{\sigma}$. Novamente utilizamos o lema `ext_o` da teoria `substitution`. Reescrevemos com este lema, utilizando a regra `rewrite`, e completamos a prova. ■

Capítulo 5

Conclusão e Trabalhos Futuros

Neste trabalho apresentamos uma formalização para a teoria de unificação de primeira ordem, desenvolvida em PVS como uma *sub-teoria* da *teoria* `trs` [8] para sistemas de reescrita de termos, denominada `unification`. A *sub-teoria* `unification` foi desenvolvida com base no algoritmo de unificação de Robinson, onde formalizamos e provamos a correção de uma versão deste algoritmo, que corresponde à existência de *mgu*'s para termos unificáveis.

Na teoria `trs` encontramos uma base sólida de conceitos, como por exemplo as definições de termo, posição, substituição, dentre outros, além de vários resultados formalizados sobre tais conceitos que foram essenciais para o desenvolvimento da *sub-teoria* `unification`.

A partir da formalização da correção do algoritmo de unificação, feita em dois teoremas onde provamos que a substituição computada pelo algoritmo de unificação, tendo como entradas dois termos unificáveis, é um elemento do conjunto de unificadores destes dois termos e é também mais geral que qualquer outro elemento deste conjunto, chegamos à formalização do resultado principal da *sub-teoria* `unification`, o teorema que estabelece a existência de substituições mais gerais para dois termos unificáveis, que completa a *teoria* `trs`, já que isto foi tratado anteriormente como um axioma. Em particular, na prova do teorema dos pares críticos de Knuth-Bendix é necessário, quando da aplicação de juntabilidade de divergências geradas por instâncias de pares críticos. Assim, com a conclusão da formalização deste teorema da *sub-teoria* `unification`, atingimos o objetivo inicial deste trabalho que era desenvolver uma *sub-teoria* para tratar da existência e unicidade de unificadores mais gerais dentro da teoria `trs`.

Na *sub-teoria* `unification` foram formalizados 27 lemas. Alguns lemas auxiliares foram acrescentados às *sub-teorias* `substitution`, `subterm` e `position`, totalizando 14 lemas. Em seguida apresentamos na Tabela 5.0.1 alguns dados quantitativos acerca da *sub-teoria* `unification`. Estes dados consistem de informações sobre a quantidade de linhas de especificação e de prova e sobre o tamanho da especificação e do arquivo de provas.

Tabela 5.0.1: Análise Quantitativa da *sub-teoria* `unification`

Arquivos	Linhas	Tamanho
arquivo de especificação	274	9.6 KB
arquivo de provas	11404	637.4 KB

Além dos lemas especificados, tivemos um total de 36 TCC's gerados, isto é, 36 obrigações de prova geradas automaticamente durante a checagem de tipos realizada pelo *typechecker* do PVS. Dentre o total de TCC's, 30 foram provados manualmente e 6 foram provados diretamente pelo *provador*, com alguma dependência em relação a TCC's anteriores.

Um ponto importante e bastante positivo do assistente de provas PVS, é que sua linguagem de especificação permite desenvolver teorias com uma linguagem muito próxima da encontrada nos livros texto. Assim, foi possível desenvolver a *sub-teoria* `unification` de forma que, mesmo um leitor não familiarizado com o PVS não encontre dificuldades em compreender os conceitos formalizados.

5.1 Trabalhos Relacionados

Existem outras verificações de correção de algoritmos de unificação desenvolvidas em outros assistentes de prova, por exemplo em LCF por Paulson [24], em Booyer-Moore por Kaufmann [12], em ACL2 por Ruiz-Reina *et al* [29], em Coq por Joseph Rouyer [28]. Há também uma formalização em Isabelle, desenvolvida por Konrad Slind, denominada *Unify*, e uma versão melhorada desta, desenvolvida por Alexander Krauss, também em Isabelle, denominada *Unification*.

A primeira formalização do algoritmo de unificação foi a de Paulson, onde fez-se uma verificação da teoria de Manna e Waldinger [15] para substituições e unificadores mais gerais. Na teoria de Manna e Waldinger deriva-se um algoritmo de unificação a partir da prova de que sua especificação é possível. Em contraste com a representação de termos que usamos neste trabalho, dada pela Definição 2.2.2, Paulson trata os termos como sendo ou variáveis ou constantes ou combinações de dois termos, isto é, os termos são restritos a serem binários. A estrutura dos termos é construída recursivamente em LCF da seguinte forma, onde `COMB` é um combinador binário:

$$
\begin{array}{l}
struct_axm(\text{“}: term\text{”}, \text{‘}strict\text{’}, \\
\quad [\quad \text{‘}\mathtt{CONST}\text{’}, [\text{“}c : const\text{”}]; \\
\quad\quad \text{‘}\mathtt{VAR}\text{’}, [\text{“}v : var\text{”}]; \\
\quad\quad \text{‘}\mathtt{COMB}\text{’}, [\text{“}t_1 : term\text{”}; \text{“}t_2 : term\text{”}]]);;
\end{array}
$$

A especificação da teoria *Unify* desenvolvida por Slind em Isabelle segue a mesma abordagem feita por Paulson, implementando a formalização de Manna e Waldinger com algumas simplificações novas, contudo o algoritmo ainda é especificado com base em uma estrutura de termos construídos por um combinador binário:

```
datatype 'a uterm =
    Var 'a
  | Const 'a
  | Comb "'a uterm" "'a uterm"
```

Já a teoria *Unification*, que também é uma formalização de um algoritmo de unifcação de primeira ordem, é basicamente uma versão melhorada da formalização prévia onde os termos são tratados da mesma forma que o fazemos neste trabalho.

```
datatype ‘a trm =
  Var ‘a
  | Const ‘a
  | App "‘a trm" "‘a trm" (infix "." 60)
```

Tanto na teoria *Unify* como na teoria *Unification* é definido um algoritmo de unificação e depois prova-se a sua correção. Similarmente à abordagem apresentada neste trabalho, notamos que nas especificações feitas em Isabelle, a idempotência da substituição computada pelo algoritmo de unificação não é necessária para provar que o algoritmo é correto ou que termina.

Como observado anteriormente, o algoritmo de unificação de Robinson é ineficiente, pois é exponencial em tempo de execução e em complexidade de espaço. No trabalho de Ruiz-Reina *et al*, é apresentada uma verificação de um algoritmo que usa uma estrutura de dados eficiente, onde os termos são representados como grafos acíclicos diretos, o resultado é a formalização da correção de um algoritmo de unificação quadrático em tempo de execução. Esta especificação é baseada na exposição de Corbin e Bidoit [5], onde os autores mostram que uma escolha apropriada de uma estrutura de dados para representar termos torna a complexidade do algoritmo de unificação de Robinson, que é sabidamente exponencial, quadrática.

5.2 Trabalhos Futuros

Neste trabalho apresentamos uma especificação de um algoritmo que tem como entradas dois termos unificáveis, isto é, excluímos a possibilidade de falha no processo de unificação, pois a fim de obter uma *teoria* em PVS para sistemas de reescrita de termos, era extritamente necessário mostrar a exitência de unificadores mais gerais para termos unificáveis. Assim, exercícios simples de formalização podem ser propostos, no sentido de especificar uma nova função, para representar uma versão completa do algoritmo de unificação, que seja total no conjunto de termos, pois esta que especificamos é total sobre o conjunto de termos unificáveis, onde essa restrição foi possível devido ao sistema de tipos dependentes do PVS. Neste sentido, pode-se idealizar um novo operador que segue a mesma estrutura do operador `sub_of_frst_diff`, mas detecte falhas. Para isto começa-se propondo uma adaptação no operador `resolving_diff`, que consiste em eliminar restrições sobre os parâmetros. As restrições sobre os parâmetros do operador `resolving_diff` limitam a ação deste operador a termos unificáveis e diferentes. Assim, eliminariamos a restrição que limita os parâmetros a termos unificáveis. Com isto, se a posição p da primeira diferença entre os termos s e t, obtida pelo operador modificado `resolving_diff`, é tal que uma das possibilidades ocorre:

- $s|_p$ e $t|_p$ são termos funcionais com símbolos de função principais diferentes,
- $s|_p \in \mathcal{V}$ e $s|_p \in \mathcal{V}ars(t|_p)$,

- $t|_p \in \mathcal{V}$ e $t|_p \in \mathcal{V}ars(s|_p)$,

então o operador modificado `sub_of_frst_diff` irá retornar uma substituição que poderia ser denominada *fail*. Algumas mudanças no operador `unification_algorithm` também serão necessárias, para considerar o caso em que a substituição computada por `sub_-of_frst_diff` é *fail*. Outra proposta, mais elaborada de um projeto de formalização, é desenvolver uma especificação de um algoritmo de unificação eficiente, como a de Ruiz-Reina *et-al* para ACL2.

Além disso, temos outras propriedades importantes sobre unificação e *mgu* que podem ser formalizadas na *sub-teoria* `unification`. Dentre estas temos a propriedade de idempotência, que como vimos não é necessária na demonstração de generalidade. Contudo já temos especificado o conceito de substituição idempotente bem como a formalização de uma propriedade necessária e suficiente para que uma substituição seja idempotente. Trata-se de um teorema da *sub-teoria* `substitution` onde temos que uma substituição é idempotente se, e somente se o conjunto formado pelas variáveis do domínio da substituição e o conjunto formado pelas variáveis da imagem da substituição são disjuntos. Assim, partindo deste lema pode-se formalizar um teorema que garanta a idempotência da substituição computada pelo algoritmo de unificação. Outra propriedade ainda não formalizada é a de unicidade dos *mgu*'s. Para formalizar esta propriedade precisamos da definição de renomeamento, que já está especificada na *sub-teoria* `substitution`. Essencialmente deve ser formalizado que para qualquer *mgu* $\theta \in \mathcal{U}(s,t)$, se σ é o *mgu* computado pelo algoritmo de unificação, então σ é uma variante de θ no sentido de que $\theta \lesssim \sigma$ e $\sigma \lesssim \theta$.

Apêndice A

O Código da Especificação

Neste apêndice apresentamos o código completo da especificação da *sub-teoria* `unification`, que no capítulo 4 foi apresentado por partes. Aqui é possível ter uma visão geral da *sub-teoria* `unification`, observando todos os lemas formalizados e a disposição de cada um dentro da especificação, que finaliza com o teorema 4.3.1.

```
%%-------------------** Term Rewriting System (TRS) **------------------------
%%
%% Authors         : Andreia Borges Avelar and
%%                   Mauricio Ayala Rincon
%%                   Universidade de Brasilia - Brasil
%%
%%                         and
%%
%%                   Andre Luiz Galdino
%%                   Universidade Federal de Goias - Brasil
%%
%% Last Modified On: September 29, 2009
%%
%%---------------------------------------------------------------------------

unification[variable: TYPE+, symbol: TYPE+, arity: [symbol -> nat]]: THEORY
BEGIN

   ASSUMING

     IMPORTING variables_term[variable,symbol,arity],
               sets_aux@countability[term],
               sets_aux@countable_props[term]

     var_countable: ASSUMPTION is_countably_infinite(V)

   ENDASSUMING

   IMPORTING substitution[variable,symbol, arity]
```

```
                Vs: VAR set[(V)]
            V1, V2: VAR finite_set[(V)]
                V3: VAR finite_set[term]
           x, y, z: VAR (V)
   tau, sig, sigma,
 delta, rho, theta: VAR Sub
           st, stp: VAR finseq[term]
   r, s, t, t1, t2: VAR term
                 n: VAR nat
      p, q, p1, p2: VAR position
                 R: VAR pred[[term, term]]


%%%% Defining an instance of a term %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 instance(t, s): bool = EXISTS sigma: ext(sigma)(s) = t;


%%%% Defining substitution more general "<=" %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 <=(theta, sigma): bool = EXISTS tau: sigma = comp(tau, theta)

 mg_po: LEMMA preorder?(<=)


%%%% Defining unification between two terms %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 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)}


%%%% Defining a most general unifier "mgu" %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 mgu(theta)(s,t): bool = member(theta, U(s,t)) &
                   FORALL sigma: member(sigma, U(s,t)) IMPLIES theta <= sigma


%%%% Initial auxiliary lemma %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 uni_diff_equal_length_arg : LEMMA
      FORALL (s: term, (t: term | unifiable(s, t) & s /= t), f: symbol,
             st: {args: finite_sequence[term] | args`length = arity(f)}):
      NOT st`length = 0 AND s = app(f, st) IMPLIES
      (FORALL (fp: symbol, stp: {args: finite_sequence[term] |
               args`length = arity(fp)}): t = app(fp, stp) IMPLIES
       (f = fp & st`length = stp`length))


%%%% Position of the first difference between %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% two unifiable and different terms %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
resolving_diff(s : term, (t : term | unifiable(s,t) & s /= t ) ):
  RECURSIVE position =
   (CASES s OF
      vars(s) : empty_seq,
      app(f, st) :
      IF length(st) = 0 THEN empty_seq
      ELSE
       (CASES t OF
          vars(t) : empty_seq,
          app(fp, stp) :
          LET k : below[length(stp)] =
            min({kk : below[length(stp)] |
                        subtermOF(s,#(kk+1)) /= subtermOF(t,#(kk+1))}) IN
            add_first(k+1,
                     resolving_diff(subtermOF(s,#(k+1)),subtermOF(t,#(k+1))))
        ENDCASES)
      ENDIF
    ENDCASES)
MEASURE s BY <<


%%%% Lemmas about resolving_diff %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 resol_diff_nonempty_implies_funct_terms : LEMMA
    FORALL (s: term, (t: term | unifiable(s, t) & s /= t)):
           resolving_diff(s,t) /= empty_seq IMPLIES
           (app?(s) AND app?(t))

 resol_diff_to_rest_resol_diff : LEMMA
    FORALL (s: term, (t: term | unifiable(s, t) & s /= t)):
        LET rd =  resolving_diff(s,t) IN
            rd /= empty_seq IMPLIES
            resolving_diff(subtermOF(s,#(first(rd))),
                           subtermOF(t,#(first(rd)))) = rest(rd)

 position_s_resolving_diff : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t, p : position):
        p = resolving_diff(s, t) IMPLIES positionsOF(s)(p);

 position_t_resolving_diff : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t, p : position):
        p = resolving_diff(s, t) IMPLIES positionsOF(t)(p);

 resolving_diff_has_diff_argument : LEMMA
    FORALL (s : term, t : term | unifiable(s,t) & s /= t,
            p : position | positionsOF(s)(p) & positionsOF(t)(p)):
        p = resolving_diff(s, t) IMPLIES
          subtermOF(s, p) /= subtermOF(t, p)

 resolving_diff_has_unifiable_argument : LEMMA
    FORALL (s : term, t : term | unifiable(s,t) & s /= t,
            p : position | positionsOF(s)(p) & positionsOF(t)(p)):
        p = resolving_diff(s, t) IMPLIES
          unifiable(subtermOF(s, p), subtermOF(t, p))

 resolving_diff_vars : LEMMA
```

```
   FORALL (s : term, t : term | unifiable(s, t) & s /= t,
   p : position | positionsOF(s)(p) & positionsOF(t)(p)):
       p = resolving_diff(s, t) IMPLIES
   vars?(subtermOF(s, p)) OR vars?(subtermOF(t, p))


%%%% Auxiliary lemmas about substitutions and unifiers %%%%%%%%%%%%%%%%%%%%%%

 unifier_o : LEMMA
       member(sig, U(ext(theta)(s), ext(theta)(t))) IMPLIES
       member(comp(sig, theta), U(s, t))

 mgu_o : LEMMA
       sig <= rho IMPLIES comp(sig, theta) <= comp(rho, theta)

 unifier_and_subs : LEMMA
       member(theta, U(s, t)) IMPLIES
       (FORALL (sig: Sub): member(comp(sig, theta), U(s, t)))

 idemp_mgu_iff_all_unifier : LEMMA
   FORALL (theta: Sub | member(theta, U(s, t))):
       mgu(theta)(s, t) & idempotent_sub?(theta) IFF
       (FORALL (sig: Sub | member(sig, U(s, t))): sig = comp(sig, theta))

 unifiable_terms_unifiable_args : LEMMA
   FORALL (s : term, t : term,
           p : position | positionsOF(s)(p) & positionsOF(t)(p)):
       member(sig, U(s, t)) IMPLIES
          member(sig, U(subtermOF(s, p), subtermOF(t, p)))

 var_term_unifiable_not_var_in_term : LEMMA
   FORALL (s : term, t : term ):
       vars?(s) & unifiable(s, t) & s /= t IMPLIES
          NOT member(s, Vars(t))


%%%% Substitution to fix the %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% first difference %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 sub_of_frst_diff(s : term , (t : term | unifiable(s,t) & s /= t )) : Sub =
  LET k : position = resolving_diff(s,t) IN
     LET sp = subtermOF(s,k) , tp = subtermOF(t,k) IN
       IF vars?(sp)
       THEN (LAMBDA (x : (V)) : IF x = sp THEN tp ELSE x ENDIF)
       ELSE (LAMBDA (x : (V)) : IF x = tp THEN sp ELSE x ENDIF)
       ENDIF


%%%% Lemmas about "sub_of_frst_diff" %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 dom_sub_of_frst_diff_is : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t, sig : Sub):
       sig = sub_of_frst_diff(s, t) AND p = resolving_diff(s, t)
       IMPLIES
          IF vars?(subtermOF(s, p))
            THEN Dom(sig) = singleton(subtermOF(s, p))
```

```
            ELSE Dom(sig) = singleton(subtermOF(t, p))
          ENDIF

 var_sub_1stdiff_not_member_term : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s ,t) IN
           FORALL ( x | member(x,Dom(sig)), r | member(r,Ran(sig) )) :
               NOT member(x, Vars(r))

 sub_of_frst_diff_unifier_o : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t):
        member(rho, U(s, t)) IMPLIES
           LET sig = sub_of_frst_diff(s, t) IN
              EXISTS theta : rho = comp(theta, sig)

 ext_sub_of_frst_diff_unifiable : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
           unifiable(ext(sig)(s), (ext(sig)(t)))

 sub_of_frst_diff_remove_x : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
           Dom(sig)(x) IMPLIES
               (NOT member(x, Vars(ext(sig)(s)))) AND
               (NOT member(x, Vars(ext(sig)(t))))

 vars_sub_of_frst_diff_s_is_subset_union : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
           subset?(Vars(ext(sig)(s)), union( Vars(s), Vars(t)))

 vars_sub_of_frst_diff_t_is_subset_union : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
           subset?(Vars(ext(sig)(t)), union( Vars(s), Vars(t)))

 union_vars_ext_sub_of_frst_diff : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t) :
        LET sig = sub_of_frst_diff(s, t) IN
           union(Vars(ext(sig)(s)), Vars(ext(sig)(t)))
           = difference(union( Vars(s), Vars(t)), Dom(sig))

 vars_ext_sub_of_frst_diff_decrease : LEMMA
    FORALL (s : term, t : term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
           Card(union( Vars(ext(sig)(s)), Vars(ext(sig)(t))))
           < Card(union( Vars(s), Vars(t)))


%%%% Function to compute a unifier of %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% two unifiable terms %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 unification_algorithm(s : term, (t : term | unifiable(s,t))) : RECURSIVE Sub =
    IF s = t THEN identity
    ELSE LET sig = sub_of_frst_diff(s, t) IN
```

```
          comp( unification_algorithm(ext(sig)(s) , ext(sig)(t)) , sig)
   ENDIF
 MEASURE Card(union(Vars(s), Vars(t)))


%%%% Lemmas about "unification_algorithm" %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 unification_algorithm_gives_unifier : LEMMA
   unifiable(s,t) IMPLIES member(unification_algorithm(s, t), U(s, t))

 unification_algorithm_gives_mg_subs : LEMMA
   member(rho, U(s, t)) IMPLIES unification_algorithm(s, t) <= rho


%%%% Existence of a most general unifier %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 unification : LEMMA
   unifiable(s,t) => EXISTS theta : mgu(theta)(s,t)


END unification
```

Apêndice B

Formalização do lema `sub_of_frst_diff_remove_x`

Apresenta-se a formalização do lema `sub_of_frst_diff_remove_x`, com o intuito de apresentar uma formalização completa, explicando cada regra de prova utilizada.

Neste lema formalizamos o fato de que, se s e t são dois termos unificáveis e se σ é a substituição que resolve a primeira diferença entre os termos s e t então, se x é uma variável do domínio de σ, então x não é membro de $\mathcal{V}ars(\hat{\sigma}(s))$ e nem de $\mathcal{V}ars(\hat{\sigma}(t))$. De fato, sabemos que σ substitui todas as ocorrências de x, em um termo instanciado por σ, por $\sigma(x)$.

Primeiro apresentamos a especificação deste lema, que também pode ser encontrada no apêndice A.

```
sub_of_frst_diff_remove_x : LEMMA
   FORALL (s : term, t : term | unifiable(s, t) & s /= t):
       LET sig = sub_of_frst_diff(s, t) IN
          Dom(sig)(x) IMPLIES
              (NOT member(x, Vars(ext(sig)(s)))) AND
              (NOT member(x, Vars(ext(sig)(t))))
```

Durante a apresetação da formalização, procederemos da seguinte forma: exporemos um sequente e em seguida a regra aplicada a este sequente assim como vemos em PVS, em seguida uma breve explicação da regra aplicada.

Assim, começamos com o seguinte sequente:

```
  |-------
{1}   FORALL (x: (V), s: term, t: term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
          Dom(sig)(x) IMPLIES
           (NOT member(x, Vars(ext(sig)(s)))) AND
            (NOT member(x, Vars(ext(sig)(t))))

Rule? (skosimp)
```

Com a regra `skosimp` skolemizamos as variáveis ligadas da fórmula 1. E obtemos:

```
  |-------
{1}   LET sig = sub_of_frst_diff(s!1, t!1) IN
        Dom(sig)(x!1) IMPLIES
         (NOT member(x!1, Vars(ext(sig)(s!1)))) AND
          (NOT member(x!1, Vars(ext(sig)(t!1))))

Rule? (assert)
```

Com a regra `assert`, realizamos simplificações através de procedimentos de decisão. E obtemos:

```
  |-------
{1}   Dom(sub_of_frst_diff(s!1, t!1))(x!1) IMPLIES
       (NOT member(x!1, Vars(ext(sub_of_frst_diff(s!1, t!1))(s!1)))) AND
        (NOT member(x!1, Vars(ext(sub_of_frst_diff(s!1, t!1))(t!1))))

Rule? (name-replace "sig!1" "sub_of_frst_diff(s!1, t!1)" :hide? nil)
```

Com a regra `name-replace`, apenas renomeamos e substituimos todas as ocorrências de `sub_of_frst_diff(s!1, t!1)` por `sig!1`. E obtemos:

```
{-1}  sub_of_frst_diff(s!1, t!1) = sig!1
  |-------
{1}   Dom(sig!1)(x!1) IMPLIES
       (NOT member(x!1, Vars(ext(sig!1)(s!1)))) AND
        (NOT member(x!1, Vars(ext(sig!1)(t!1))))

Rule? (prop)
```

A regra `prop` é uma regra de simplificação proposicional. O sequente acima corresponde a seguinte situção: $D \Rightarrow A \rightarrow (\neg B \vee \neg C)$, com a aplicação da regra `prop` temos uma ramificação na prova onde são gerados dois objetivos, um da forma $B, D, A \Rightarrow$ e outro da forma $C, D, A \Rightarrow$. Explicaremos em seguida apenas o primeiro ramo, o segundo é idêntico e isto pode ser observado na sua formalização que apresentamos sem comentar. Assim, temos:

```
{-1}  member(x!1, Vars(ext(sig!1)(s!1)))
[-2]  sub_of_frst_diff(s!1, t!1) = sig!1
{-3}  Dom(sig!1)(x!1)
  |-------

Rule? (lemma "vars_subst_not_in")
```

Com esta regra, apenas trazemos para o antecedente uma nova fórmula que corresponde ao lema "invocado", que neste caso é o lema `vars_subst_not_in`. E obtemos:

```
{-1}  FORALL (t: term[variable, symbol, arity],
              sigma: Sub[variable, symbol, arity], x: (V)):
        (Dom(sigma)(x) AND
          (FORALL (r: term[variable, symbol, arity]):
             Ran(sigma)(r) IMPLIES NOT member(x, Vars(r))))
         IMPLIES NOT member(x, Vars(ext(sigma)(t)))
[-2]  member(x!1, Vars(ext(sig!1)(s!1)))
[-3]  sub_of_frst_diff(s!1, t!1) = sig!1
[-4]  Dom(sig!1)(x!1)
  |-------

Rule? (inst -1 "s!1" "sig!1" "x!1")
```

Agora, instanciamos a hipótese do lema "invocado" com as devidas variáveis. E obtemos:

```
{-1}  (Dom(sig!1)(x!1) AND
        (FORALL (r: term[variable, symbol, arity]):
           Ran(sig!1)(r) IMPLIES NOT member(x!1, Vars(r))))
       IMPLIES NOT member(x!1, Vars(ext(sig!1)(s!1)))
[-2]  member(x!1, Vars(ext(sig!1)(s!1)))
[-3]  sub_of_frst_diff(s!1, t!1) = sig!1
[-4]  Dom(sig!1)(x!1)
  |-------

Rule? (assert)
```

Novamente realizamos simplificações com a regra `assert`. E obtemos:

```
[-1]  member(x!1, Vars(ext(sig!1)(s!1)))
[-2]  sub_of_frst_diff(s!1, t!1) = sig!1
[-3]  Dom(sig!1)(x!1)
  |-------
{1}   FORALL (r: term[variable, symbol, arity]):
        Ran(sig!1)(r) IMPLIES NOT member(x!1, Vars(r))

Rule? (skosimp)
```

Novamente skolemizamos as variáveis ligadas da fórmula 1. E obtemos:

```
{-1}  Ran(sig!1)(r!1)
{-2}  member(x!1, Vars(r!1))
[-3]  member(x!1, Vars(ext(sig!1)(s!1)))
[-4]  sub_of_frst_diff(s!1, t!1) = sig!1
[-5]  Dom(sig!1)(x!1)
  |-------

Rule? (hide -3)
```

Escondemos a fórmula -3, pois neste momento ela não é necessária na derivação, porém continua na memória do provador, caso desejemos reavê-la. Assim, temos:

```
[-1]  Ran(sig!1)(r!1)
[-2]  member(x!1, Vars(r!1))
[-3]  sub_of_frst_diff(s!1, t!1) = sig!1
[-4]  Dom(sig!1)(x!1)
  |-------

Rule? (lemma "var_sub_1stdiff_not_member_term")
```

Novamente usamos a regra `lema` para trazer às fórmulas do antecedente o lema chamado, que no caso é o lema `var_sub_1stdiff_not_member_term`. E obtemos:

```
{-1}  FORALL (s: term, t: term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
          FORALL (x: (V) | member(x, Dom(sig)),
                  r: term[variable, symbol, arity] | member(r, Ran(sig))):
            NOT member(x, Vars(r))
[-2]  Ran(sig!1)(r!1)
[-3]  member(x!1, Vars(r!1))
[-4]  sub_of_frst_diff(s!1, t!1) = sig!1
[-5]  Dom(sig!1)(x!1)
  |-------

Rule? (inst -1 "s!1" "t!1")
```

Instanciamos a fórmula -1 com as devidas variáveis. E obtemos:

```
{-1}  LET sig = sub_of_frst_diff(s!1, t!1) IN
        FORALL (x: (V) | member(x, Dom(sig)),
                r: term[variable, symbol, arity] | member(r, Ran(sig))):
          NOT member(x, Vars(r))
[-2]  Ran(sig!1)(r!1)
[-3]  member(x!1, Vars(r!1))
[-4]  sub_of_frst_diff(s!1, t!1) = sig!1
[-5]  Dom(sig!1)(x!1)
  |-------

Rule? (replaces -4)
```

Substituimos todas as ocorrências de `sub_of_frst_diff(s!1, t!1)` em todas as fórmulas do sequente, por `sig!1`, e escondemos a fórmula -4. Com isto, obtemos:

```
{-1}  LET sig = sig!1 IN
        FORALL (x: (V) | member(x, Dom(sig)),
                r: term[variable, symbol, arity] | member(r, Ran(sig))):
          NOT member(x, Vars(r))
{-2}  Ran(sig!1)(r!1)
{-3}  member(x!1, Vars(r!1))
{-4}  Dom(sig!1)(x!1)
  |-------

Rule? (assert)
```

Novamente realizamos simplificações com a regra `assert`. E obtemos:

```
{-1}  FORALL (x: (V) | member(x, Dom(sig!1)),
              r: term[variable, symbol, arity] | member(r, Ran(sig!1))):
        NOT member(x, Vars(r))
[-2]  Ran(sig!1)(r!1)
[-3]  member(x!1, Vars(r!1))
[-4]  Dom(sig!1)(x!1)
  |-------

Rule? (inst -1 "x!1" "r!1")
```

Istanciamos a fórmula -1 com as devidas variáveis. Neste ponto da prova a operação de checagem de tipos gera outro subobjetivo. Mas antes temos:

```
[-1]  Ran(sig!1)(r!1)
[-2]  member(x!1, Vars(r!1))
[-3]  Dom(sig!1)(x!1)
  |-------
{1}   member[term[variable, symbol, arity]]
          (r!1, Ran[variable, symbol, arity](sig!1))

Rule? (hide -2 -3)
```

Escondemos as fórmulas -2 e -3 e obtemos:

```
[-1]  Ran(sig!1)(r!1)
  |-------
[1]   member[term[variable, symbol, arity]]
          (r!1, Ran[variable, symbol, arity](sig!1))

Rule? (expand "member")
```

Expandimos a definição de `member` e com isto obtemos:

```
[-1]  Ran(sig!1)(r!1)
  |-------
{1}   Ran[variable, symbol, arity](sig!1)(r!1)
```

Neste ponto obtemos uma contradição, o que nos leva a completar este ramo da prova. Assim, passamos ao objetivo seguinte onde temos:

```
[-1]  Ran(sig!1)(r!1)
[-2]  member(x!1, Vars(r!1))
[-3]  Dom(sig!1)(x!1)
  |-------
{1}   member[(V)](x!1, Dom[variable, symbol, arity](sig!1))

Rule? (hide -1 -2)
```

Escondemos as fórmulas -1 e -2 e passamos ao sequente:

```
[-1]  Dom(sig!1)(x!1)
  |-------
[1]   member[(V)](x!1, Dom[variable, symbol, arity](sig!1))

Rule? (expand "member")
```

Novamente expandimos a definição de `member` e obtemos:

```
[-1]  Dom(sig!1)(x!1)
  |-------
{1}   Dom[variable, symbol, arity](sig!1)(x!1)
```

Onde temos uma contradição. Com isto completamos este ramo da prova e passamos ao objetivo seguinte.

Como comentamos anteriormente, o próximo ramo da prova é verificado através da mesma sequência de regras de provas que acabamos de descrever. Assim, mostramos a prova deste ramo, mas sem comentar novamente cada regra.

```
{-1}  member(x!1, Vars(ext(sig!1)(t!1)))
[-2]  sub_of_frst_diff(s!1, t!1) = sig!1
{-3}  Dom(sig!1)(x!1)
  |-------

Rule? (lemma "vars_subst_not_in")


{-1}  FORALL (t: term[variable, symbol, arity],
              sigma: Sub[variable, symbol, arity], x: (V)):
        (Dom(sigma)(x) AND
          (FORALL (r: term[variable, symbol, arity]):
             Ran(sigma)(r) IMPLIES NOT member(x, Vars(r))))
         IMPLIES NOT member(x, Vars(ext(sigma)(t)))
[-2]  member(x!1, Vars(ext(sig!1)(t!1)))
[-3]  sub_of_frst_diff(s!1, t!1) = sig!1
[-4]  Dom(sig!1)(x!1)
  |-------

Rule? (inst -1 "t!1" "sig!1" "x!1")


{-1}  (Dom(sig!1)(x!1) AND
        (FORALL (r: term[variable, symbol, arity]):
           Ran(sig!1)(r) IMPLIES NOT member(x!1, Vars(r))))
       IMPLIES NOT member(x!1, Vars(ext(sig!1)(t!1)))
[-2]  member(x!1, Vars(ext(sig!1)(t!1)))
[-3]  sub_of_frst_diff(s!1, t!1) = sig!1
[-4]  Dom(sig!1)(x!1)
  |-------

Rule? (assert)


[-1]  member(x!1, Vars(ext(sig!1)(t!1)))
[-2]  sub_of_frst_diff(s!1, t!1) = sig!1
[-3]  Dom(sig!1)(x!1)
  |-------
{1}   FORALL (r: term[variable, symbol, arity]):
        Ran(sig!1)(r) IMPLIES NOT member(x!1, Vars(r))

Rule? (skosimp)
```

```
{-1}  Ran(sig!1)(r!1)
{-2}  member(x!1, Vars(r!1))
[-3]  member(x!1, Vars(ext(sig!1)(t!1)))
[-4]  sub_of_frst_diff(s!1, t!1) = sig!1
[-5]  Dom(sig!1)(x!1)
  |-------

Rule? (hide -3)


[-1]  Ran(sig!1)(r!1)
[-2]  member(x!1, Vars(r!1))
[-3]  sub_of_frst_diff(s!1, t!1) = sig!1
[-4]  Dom(sig!1)(x!1)
  |-------

Rule? (lemma "var_sub_1stdiff_not_member_term")


{-1}  FORALL (s: term, t: term | unifiable(s, t) & s /= t):
        LET sig = sub_of_frst_diff(s, t) IN
          FORALL (x: (V) | member(x, Dom(sig)),
                  r: term[variable, symbol, arity] | member(r, Ran(sig))):
            NOT member(x, Vars(r))
[-2]  Ran(sig!1)(r!1)
[-3]  member(x!1, Vars(r!1))
[-4]  sub_of_frst_diff(s!1, t!1) = sig!1
[-5]  Dom(sig!1)(x!1)
  |-------

Rule? (inst -1 "s!1" "t!1")


{-1}  LET sig = sub_of_frst_diff(s!1, t!1) IN
        FORALL (x: (V) | member(x, Dom(sig)),
                r: term[variable, symbol, arity] | member(r, Ran(sig))):
          NOT member(x, Vars(r))
[-2]  Ran(sig!1)(r!1)
[-3]  member(x!1, Vars(r!1))
[-4]  sub_of_frst_diff(s!1, t!1) = sig!1
[-5]  Dom(sig!1)(x!1)
  |-------

Rule? (replaces -4)


{-1}  LET sig = sig!1 IN
        FORALL (x: (V) | member(x, Dom(sig)),
                r: term[variable, symbol, arity] | member(r, Ran(sig))):
          NOT member(x, Vars(r))
{-2}  Ran(sig!1)(r!1)
{-3}  member(x!1, Vars(r!1))
{-4}  Dom(sig!1)(x!1)
  |-------

Rule? (assert)
```

```
{-1}  FORALL (x: (V) | member(x, Dom(sig!1)),
              r: term[variable, symbol, arity] | member(r, Ran(sig!1))):
        NOT member(x, Vars(r))
[-2]  Ran(sig!1)(r!1)
[-3]  member(x!1, Vars(r!1))
[-4]  Dom(sig!1)(x!1)
  |-------

Rule? (inst -1 "x!1" "r!1")


[-1]  Ran(sig!1)(r!1)
[-2]  member(x!1, Vars(r!1))
[-3]  Dom(sig!1)(x!1)
  |-------
{1}   member[term[variable, symbol, arity]]
          (r!1, Ran[variable, symbol, arity](sig!1))

Rule? (hide -2 -3)


[-1]  Ran(sig!1)(r!1)
  |-------
[1]   member[term[variable, symbol, arity]]
          (r!1, Ran[variable, symbol, arity](sig!1))

Rule? (expand "member")


[-1]  Ran(sig!1)(r!1)
  |-------
{1}   Ran[variable, symbol, arity](sig!1)(r!1)

which is trivially true.


[-1]  Ran(sig!1)(r!1)
[-2]  member(x!1, Vars(r!1))
[-3]  Dom(sig!1)(x!1)
  |-------
{1}   member[(V)](x!1, Dom[variable, symbol, arity](sig!1))

Rule? (hide -1 -2)


[-1]  Dom(sig!1)(x!1)
  |-------
[1]   member[(V)](x!1, Dom[variable, symbol, arity](sig!1))

Rule? (expand "member")


[-1]  Dom(sig!1)(x!1)
  |-------
{1}   Dom[variable, symbol, arity](sig!1)(x!1)

which is trivially true.
```

Referências Bibliográficas

[1] Mauricio Ayala-Rincón. Fundamentos da Programação Lógica e Funcional - O princípio de resolução e a teoria de reescrita. Notas de aula, Quarta versão, Departamento de Matemática, Universidade de Brasília, 2008. Disponível em: `http://ayala.mat.unb.br/`.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[3] Franz Baader and Wayne Snyder. Unification Theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 447–533. Elsevier Science Publishers, 2001.

[4] Robert S. Boyer and J. Strother Moore. The Sharing of Structure in Theorem-Proving Programs. In *Machine Intelligence 7*, pages 101–116. University Press, 1972.

[5] Jacques Corbin and Michel Bidoit. A Rehabilitation of Robinson's Unification Algorithm. In *IFIP Congress*, pages 909–914, 1983.

[6] André Luiz Galdino. *Uma Formalização da Teoria de Reescrita em Linguagem de Ordem Superior*. PhD thesis, Universidade de Brasília, 2008.

[7] André Luiz Galdino and Mauricio Ayala-Rincón. A Theory for Abstract Rewriting Systems in PVS. *CLEI - electronic journal*, 11(2, paper 4), 2008.

[8] André Luiz Galdino and Mauricio Ayala-Rincón. A PVS *Theory* for Term Rewriting Systems. *Electronic Notes in Theoretical Computer Science*, 247:67–83, 2009.

[9] James R. Guard. Automated Logic for Semi-Automated Mathematics. *AFCRL*, Scientific Report 1:64–411, March 1964. Disponível em: `http://handle.dtic.mil/100.2/AD602710`.

[10] Jacques Herbrand. *Recherches sur la Théorie de la Démonstration.* Ph.d. thesis, University of Paris, November 1971.

[11] Gérard Huet. *Resolution D'Equations Dans Les Langages D'Ordre 1, 2,..., ω*. PhD thesis, University of Paris, 1976.

[12] 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.

[13] Kevin Knight. Unification: A Multidisciplinary Survey. *ACM Computing Surveys*, 21(1), March 1989.

[14] Donald E. Knuth and Peter B. Bendix. Simple Word Problems in Universal Algebra. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[15] Zohar Manna and Richard Waldinger. Deductive Synthesis of the Unification Algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[16] Alberto Martelli and Ugo Montanari. Unification in Linear Time and Space: A Structured Presentation. Technical Report Internal Report $N^0B76-16$, Ist. di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, 1976.

[17] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

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

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

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

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

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

[23] Mike S. Paterson and Mark N. Wegman. Linear Unification. In *Proceedings of the Symposium on the Theory of Computing*. ACM Special Interest Group for Automata and Computability Theory (SIGACT), 1976. http://sigact.acm.org.

[24] Lawrence C. Paulson. Verifying the Unification Algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, 1985.

[25] John C. Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. In *Machine Intelligence 5*, pages 135–151. Edinburgh University Press, 1970.

[26] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[27] John Alan Robinson. Computational Logic: The Unification Computation. *Machine Intelligence*, 6:63–72, 1971.

[28] Joseph Rouyer. Développement de l'Algorithme d'Unification dans le Calcul des Constructions. Technical Report 1795, INRIA, November 1997.

[29] José-Luis Ruiz-Reina, Francisco-Jesús Martín-Mateos, José-Antonio Alonso, and María-José Hidalgo. Formal Correctness of a Quadratic Unification Algorithm. *Journal of Automated Reasoning*, 37(1-2):67–92, 2006.

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

[31] Marisa Venturini-Zilli. Complexity of the Unification Algorithm for First-Order Expressions. *Calcolo*, 12(4):361–371, December 1975.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo