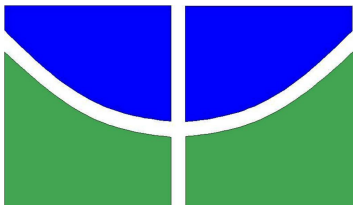


Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Matemática

Formalização da confluência para sistemas de reescrita ortogonais

por

Ana Cristina Rocha Oliveira

Brasília

2012

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Matemática

Formalização da confluência para sistemas de reescrita ortogonais

por

Ana Cristina Rocha Oliveira *

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

MESTRE EM MATEMÁTICA

agosto de 2012

Comissão Examinadora:

Prof. Dr. Mauricio Ayala-Rincón - UnB - Orientador

Prof. Dr. André Luiz Galdino - UFG - Membro

Prof. Dr. Edward Hermann Haeusler - PUC-Rio - Membro

*A autora foi bolsista do CNPq durante a elaboração deste trabalho.

Dedicatória

A *Jesus*

porque dele, por ele e para ele

são todas as coisas.

Glória, pois, a ele eternamente.

Agradecimentos

A Deus, primeiramente, por todas as portas abertas e consolo quando as coisas pareciam nebulosas e difíceis de enfrentar.

Aos meus pais Francisco de Assis e Sebastiana e ao meu irmão Bruno pelo carinho e amor em qualquer situação. Vocês foram meu grande exemplo de força e disposição em fazer o melhor.

Ao meu marido e grande amor Ricardo pelo cuidado e palavras de incentivo, a quem também dispenso o meu cuidado e afeto eternamente.

Aos meus amigos e colegas de mestrado e graduação Mayra, Gabriella, Aristóteles, Dhiego, Angélica, Getúlio, João Paulo, Robson e Linniker que me acompanharam sempre, seja na hora dos passeios e diversão, seja nas muitas horas de estudo para as provas de qualificação.

Aos meus amigos e colegas do Grupo de Teoria da Computação Andréia, Thaynara, Daniel e Kaliana por me receberem no grupo e darem todo o suporte necessário. Obrigada pela amizade. Agradeço especialmente à Andréia pela paciência em sanar as minhas muitas dúvidas neste longo processo.

Ao meu querido orientador Mauricio, que suportou toda a minha temosia e orientou-me por tantos anos com a perseverança de quem ensina a uma filha.

Aos membros da banca examinadora Edward Hermann Haeusler, Flávio Leonardo Cavalcanti de Moura e André Luiz Galdino por aceitarem o convite de compor a banca, além das críticas e sugestões que enriquecem este trabalho. Agradeço particularmente ao André, a quem recorri muitas vezes para sanar dúvidas, pela prontidão em me atender sempre.

Ao professor Celius, pelas horas de boa conversa e incentivo, além da disposição em ajudar em qualquer dificuldade. Você é um grande amigo!

Aos funcionários do MAT pela gentileza e carinho dispensados. Obrigada especialmente à Cláudia e D. Irene pelas boas conversas no subsolo.

Ao CNPq pelo apoio financeiro.

Novamente, a Jesus, que é o início e fim de todas as coisas. A minha vida e trabalho são teus, Senhor!

Resumo

Ortogonalidade é uma característica da programação que consiste, de uma maneira sintática, em garantir o determinismo de especificações funcionais. Essencialmente, a ortogonalidade não permite, por um lado, a ambiguidade inerente do não determinismo, isto é, a existência de diferentes regras que especificam a mesma função e que podem ser aplicadas simultaneamente (não ambiguidade) e, por outro, também proíbe a repetição de variáveis no lado esquerdo dessas regras (linearidade à esquerda). Na teoria dos Sistemas de Reescrita de Termos (TRSs), determinismo é identificado pela renomada propriedade de confluência, que basicamente afirma que sempre que houver possibilidades de simplificações ou computações diferentes de um termo, as respostas computadas ou os termos reduzidos obtidos devem coincidir. Embora a prova seja tecnicamente elaborada, confluência é bem conhecida como uma consequência da ortogonalidade. Dessa forma, ortogonalidade é uma importante característica matemática intrínseca à especificação de funções recursivas, sendo naturalmente aplicada em programação e especificação funcionais. A começar pela formalização da teoria de TRSs no assistente de provas PVS, esse trabalho descreve como a confluência de TRSs ortogonais está sendo formalizada utilizando essa ferramenta. Progressos substanciais foram constatados nessa pesquisa, obtendo-se até o presente momento formalizações completas para propriedades similares, porém com restrições, tais como a formalização completa para a propriedade de confluência de TRS's não ambíguos e lineares (à esquerda e à direita).
Palavras-chave: TRS, ortogonalidade, confluência, PVS, formalização.

Abstract

Orthogonality is a discipline of programming that in a syntactic manner guarantees determinism of functional specifications. Essentially, orthogonality avoids, on the one side, the inherent ambiguity of non determinism, prohibiting the existence of different rules that specify the same function and that may apply simultaneously (non-ambiguity), and, on the other side, it eliminates the possibility of occurrence of repetitions of variables in the left-hand side of these rules (left linearity). In the theory of term rewriting systems (TRSs) determinism is captured by the well-known property of confluence, that basically states that whenever different computations or simplifications from a term are possible, the computed answers or the obtained reduced terms should coincide. Although the proof is technically elaborated, confluence is well-known to be a consequence of orthogonality. Thus, orthogonality is an important mathematical discipline intrinsic to the specification of recursive functions that is naturally applied in functional programming and specification. Starting from a formalization of the theory of TRSs in the proof assistant PVS, this work describes how confluence of orthogonal TRSs is being formalized in this proof assistant. Substantial progress has been done in this research, obtaining until now complete formalizations for some similar, but restricted properties, such as a complete formalization for the property of confluence of non ambiguous and (left and right) linear TRSs.
Keywords: TRS, orthogonality, confluence, PVS, formalization.

Sumário

Introdução

0.1 Considerações Preliminares

A programação funcional, muito explorada no meio acadêmico, tem rompido fronteiras e, atualmente, é inclusive utilizada em sistemas comerciais. Nesse contexto, a reescrita também tem se estabelecido como efetivo modo de programar. Nos últimos anos, alguns trabalhos relacionados à formalização de várias propriedades de Sistemas de Reescrita de Termos (TRS's) foram desenvolvidos em [2, 5–7] e disponibilizados em [1]. Esse tipo de desenvolvimento, com natureza semelhante à do trabalho que se segue, é de grande relevância uma vez que garante características importantes, desejáveis e nem sempre óbvias a partir de outras que são facilmente detectáveis computacionalmente.

Assim, o resultado que nos propusemos a formalizar é o de que TRS's ortogonais são confluentes. Ortogonalidade diz respeito a programas cujas regras não entram em conflito, isto é, que não se sobrepõem e também tais que as premissas de cada regra de reescrita são lineares (não há repetição de variáveis do lado esquerdo das regras). Já a confluência diz respeito à independência de caminhos tomados, sendo que, para cada duas rotas divergentes, existem dois caminhos que levam à computação de um resultado comum.

Portanto, assim como a terminação é uma característica fundamental e muitas vezes buscada por obrigar a existência de formas normais (termos irredutíveis dentro de um programa), a confluência é desejada em muitos sistemas computacionais uma vez que garante a unicidade de resposta, independentemente do caminho tomado dentro do programa.

Ortogonalidade é apenas uma das propriedades capazes de garantir confluência. Anteriormente, um dos resultados que nos propusemos a formalizar foi o de que TRS's não ambíguos (sem sobreposição de regras) e lineares (sem repetição de variáveis tanto à direita quanto à esquerda das regras) são confluentes. Esse é um enfraquecimento do teorema que temos como objetivo e sua formalização já foi mencionada em [9] tendo sua conclusão ocorrida durante o tempo de decorrência do presente trabalho.

O Lema de Newman é outro resultado que garante confluência, mostrando que terminação e confluência local também implicam em confluência. A formalização deste último lema está disponível em [5] e depende exclusivamente da teoria `ars`, que trata dos Sistemas Abstratos de Reescrita (ARS's). Lemas como esses envolvendo apenas ARS's são interessantes em formalizações sobre TRS's pois é muito mais simples provar propriedades locais como confluência local, confluência forte e propriedade diamante do que a confluência. Isso ocorre devido ao tratamento da divergência em um número qualquer de passos nesta última propriedade, enquanto que naquelas três primeiras tratam-se divergências em um único passo. Portanto, provando propriedades locais obtemos a confluência através desse tipo de lema.

0.2 Objetivo

O objetivo principal do presente trabalho é apresentar o que se tem desenvolvido no sentido de formalizar o teorema de confluência para TRS's ortogonais no assistente de provas *Prototype Verification System* (PVS). Para tanto, mostraremos a prova analítica do teorema principal apresentando os pontos chave da demonstração e como usamos esses pontos no sentido de especificar e formalizar o nosso resultado. Também apresentaremos a formalização de alguns dos lemas auxiliares que foram usados no caminho da demonstração.

0.3 Contribuições

A teoria `orthogonality`, criada para tratar a formalização do teorema sobre confluência de TRS's ortogonais, é uma extensão das teorias `ars` (em [8]) e `trs` (em [6]), sendo os resultados destas teorias a base completa para o trabalho que fizemos. Esse arcabouço constitui-se da formalização de questões e propriedades apresentadas nos livros-texto ([3, 4]) envolvendo confluência, noetherianidade e TRS's em geral.

Um resultado importante, que utilizou esse arcabouço sobre ARS's e TRS's no PVS, foi o de que que TRS's não ambíguos e lineares são confluentes, tratando-se de uma formalização um pouco menos complexa do que a do teorema de confluência para TRS's ortogonais. Esse resultado foi apresentado em [9] com sua formalização completa desde 2010.

Para provar que ortogonalidade implica em confluência foi necessário especificar a redução em paralelo (denota-se esta como $\rightrightarrows$ e a redução simples como $\rightarrow$) e demonstraram-se alguns lemas importantes sobre $\rightrightarrows$ como, por exemplo:

1. $\rightarrow \subseteq \rightrightarrows$;

2. $\rightrightarrows \subseteq \rightarrow^*$, onde $\rightarrow^*$ é o fecho reflexivo-transitivo da relação;

3. $\rightrightarrows^* = \rightarrow^*$;

4. $\rightrightarrows$ tem a propriedade diamante, o que implica que também é confluente;

5. finalmente, $\rightarrow$ é confluente.

Para provar os itens 2 e 3, utilizamos muito do que já tínhamos da teoria sobre TRS's e foi necessário criar vários outros lemas técnicos já que $\rightrightarrows$ ainda não havia sido tratada pela teoria `trs`.

Logo, destacamos que as principais contribuições deste trabalho são as formalizações em PVS do lema de confluência para TRS's não ambíguos e lineares, do teorema de confluência de TRS's ortogonais e dos lemas auxiliares utilizados. Também destacamos a descrição construtiva do termo de juntabilidade necessário à prova, como veremos.

0.4 Organização do Trabalho

Começaremos o trabalho mencionando as principais regras do cálculo de sequentes dentro da Teoria de Prova, o qual é base para o assistente de prova PVS. Em seguida, estaremos a par das definições necessárias ao entendimento da Teoria de Reescrita considerada aqui, levando-se em conta propriedades dos Sistemas Abstratos de Reescrita e, mais especificamente, dos Sistemas de Reescrita de Termos. Tendo bem conhecidos os conceitos utilizados, demonstraremos analiticamente os resultados chave. Por fim, entraremos na parte voltada ao desenvolvimento da teoria no assistente de prova PVS, tratando primeiramente da especificação e, em seguida, da formalização e métodos utilizados.

Capítulo

1

Fundamentos

A intenção deste capítulo é apresentar os conceitos, as principais demonstrações analíticas e os mecanismos de prova utilizados durante a formalização do teorema de confluência de Sistemas de Reescrita de Termos (TRS's) ortogonais.

1.1 Cálculo de Sequentes no PVS

O cálculo de sequentes foi proposto na década de 30 por Gentzen dentro da Teoria de Prova com o objetivo de provar a consistência da aritmética de Peano, uma vez que enfrentava dificuldades com o método de dedução natural (também desenvolvido por Gentzen). Esse cálculo é utilizado na construção de provas e, apesar de ser pouco natural, torna as demonstrações mais simples tecnicamente.

Um sequente é uma estrutura da forma $\Sigma \vdash_{\Gamma} \Lambda$, onde Σ é o conjunto de fórmulas antecedentes, Λ é o conjunto de fórmulas consequentes e Γ é o contexto. Nós assumimos a conjunção das fórmulas em Σ, implicando na disjunção das fórmulas em Λ.

Para construir provas, nós utilizamos as regras de inferência, sendo o nosso objetivo encontrar sequentes onde é possível aplicar as regras axiomáticas que trataremos a seguir. A forma de apresentação das regras é

$$\frac{\text{premissa}}{\text{conclusão}} \text{ NOME DA REGRA}$$

Neste trabalho, as formalizações foram todas feitas no *Prototype Verification System* (PVS), um assistente de prova semiautomático que permite especificar teorias e demonstrar teoremas. O PVS está baseado numa lógica de ordem superior, isto é, que possibilita que funções sejam aplicadas a indivíduos tanto quanto a funções, levando-se em conta determinadas restrições para que não haja contradições. A semântica da lógica de ordem superior é dada pelo mapeamento de tipos bem formados da lógica em conjuntos e de termos bem formados da lógica em elementos dos conjuntos representantes dos seus

tipos.

No PVS, abre-se cada teorema num ambiente de demonstração que organiza as proposições como negativas quando as tomamos como premissas e positivas quando são teses. Assim, os sequentes no PVS são apresentados da seguinte forma:

```
[-1]  p1
[-2]  p2
[-3]  p3
      ...
  |-------
[1]   t1
[2]   t2
[3]   t3
      ...
```

Dessa forma, assume-se a conjunção dos itens negativos e busca-se provar a disjunção da parte positiva, ou seja, assumindo todas as premissas, precisa-se provar alguma tese.

A seguir, nós mostraremos algumas regras de inferência do cálculo de sequentes e sua correspondência com os comandos de prova de PVS. É importante ressaltar que o cálculo de sequentes sofreu alterações desde a sua concepção por Gentzen. Os sequentes em PVS serão mais explorados posteriormente no Capítulo 3, onde serão mostrados alguns passos da formalização.

1.1.1 Regras estruturais

As regras estruturais permitem ao sequente ser rearranjado ou enfraquecido pela introdução de novas fórmulas tanto em meio às antecedentes quanto nas consequentes. A primeira regra é a de enfraquecimento e as outras são, de certa forma, casos particulares dessa.

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

Isso significa que, se conseguimos provar alguma fórmula em Λ_1 através das fórmulas em Σ_1, continuaremos provando com Λ_2 e Σ_2 devido às inclusões.

No entanto, o comando de prova do PVS correspondente é o `(hide)`, que esconde as fórmulas desnecessárias para a prova. Assim, conseguindo uma prova de um sequente mais forte, o mais fraco será trivial e teremos a prova do nosso objetivo inicial. Dessa forma, todos os comando do PVS trabalharão no sentido de baixo para cima em relação a essas regras de inferência, com o objetivo de alcançar axiomas.

As regras de contração (**C**) e de comutatividade (**X**) são casos particulares do enfraquecimento, pois vale a relação de inclusão requerida no caso anterior.

$$\frac{a, a, \Sigma \vdash_\Gamma \Lambda}{a, \Sigma \vdash_\Gamma \Lambda}\,\mathbf{C} \qquad\qquad \frac{\Sigma \vdash_\Gamma a, a, \Lambda}{\Sigma \vdash_\Gamma a, \Lambda}\,\mathbf{C}$$

O comando de prova correspondente a **C** é `(copy)`.

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

Por outro lado, a regra de troca de ordem das proposições (**X**) não foi utilizada em PVS durante a elaboração da formalização.

1.1.2 Regra de Corte

A regra de corte representa no PVS a consideração de um caso através do comando `(case)`. A prova se divide em duas partes em que uma fórmula a é introduzida em meio às antecedentes e, na outra, nas consequentes o que quer dizer que podemos considerar a desde que provemos a. Outra forma de ver é que vale a ou $\neg a$.

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

1.1.3 Axiomas

A regra axiomática (**Ax**) faz com que o programa se dê conta de que a segue de a. Assim, com o comando `(assert)`, conseguimos fechar um ramo de prova do PVS.

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

Outras duas regras axiomáticas do cálculo de sequentes são `FALSE`$\vdash$, em que alguma fórmula antecedente é falsa, e $\vdash$`TRUE`, em que uma fórmula consequente é verdadeira.

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

1.1.4 Regras Condicionais

Como trabalhou-se com várias definições recursivas (veremos isso mais à frente), é interessante verificar a regra que controla a eliminação de `IF-THEN-ELSE` em uma prova. Consideremos $\texttt{IF}(a, b, c)$ como sendo `IF` a `THEN` b `ELSE` c. Então, utilizando comandos como `(prop)`, podemos dividir a prova nos casos em que vale a (com valor booleano) e em que não vale a.

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

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

Caso os valores de a, b e c não sejam booleanos, nós podemos aplicar o comando `(lift-if)` para que `IF-THEN-ELSE` seja colocado mais exteriormente na fórmula.

1.1.5 Regras de Igualdade

A notação $a[e]$ utilizada nas regras abaixo marca uma ou mais ocorrências de e em a tal que não haja variáveis livres em e. O mesmo vale para o conjunto de fórmulas $\Lambda[e]$.

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

Para utilização da regra **Repl** utilizamos o comando `(replace i)`, onde `i` é o índice da fórmula $a = b$ no sequente do PVS.

Nós temos alguns casos especiais para replace em que uma fórmula a antecedente pode ser tratada como $a =$`TRUE` e, quando a é uma fórmula consequente, pode ser tratada como $a =$`FALSE`.

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

Por fim, temos a regra `TRUE-FALSE` que dá conta que `TRUE` e `FALSE` têm valores distintos.

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

1.1.6 Regras de Extensionalidade

As regras de extensionalidade são também regras de igualdade para estabalecer a equivalência entre duas expressões de funções ou produtos. A regra **FunExt** introduz uma constante de skolemização s (s não pertencente ao contexto Γ, isto é, $\Gamma(s)$ é indefinido) para determinar que duas funções f e g são equivalentes sempre que a aplicação das duas a um argumento arbitrário s produz resultados iguais.

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

Assim, vemos que uma igualdade entre funções de tipo $A \to B$ se transforma numa igualdade entre elementos de tipo B.

A regra **TupExt** dá conta que dois produtos são iguais se suas projeções correspondentes o são.

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

O comando do PVS correspondente a essas regras é o `(decompose-equality)`.

1.1.7 Regra de Apresentação de Tipo

A regra **Typepred** é responsável por introduzir uma fórmula mostrando que um determinado elemento tem tipo A no constexto Γ.

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

De fato, analisando a primeira linha da regra, vemos que é extraído do contexto Γ que a tem tipo A ($\tau(\Gamma)(a) = A$) e acrescenta-se a proposição correspondente nas premissas do sequente ($\pi(A)(a), \Sigma \vdash_\Gamma \Lambda$).

Nesse caso, o comando do PVS responsável pela aplicação dessa regra é o próprio (`typepred` a).

1.2 Teoria de Reescrita

A teoria de reescrita, tomada como importante paradigma dentro da programação funcional, pode apresentar diversas peculiaridades, desde que exibam sempre pares de elementos relacionados num determinado contexto. Nesse sentido, voltamos o nosso olhar para os Sistemas de Reescrita de Termos (TRS's), mas antes veremos como tratar características comuns a quaisquer sistemas de reescrita.

1.2.1 Sistemas Abstratos de Reescrita

Sistemas Abstratos de Reescrita (ARS's) são formados basicamente por um conjunto A com uma relação binária $\to_R$ sobre o conjunto, sendo denotado por $(A, \to_R)$. Dizemos que, se $a, b \in A$ e $a \to_R b$, a está relacionado com b por $\to_R$, ou a **reescreve** para b por $\to_R$, ou a **reduz** para b por $\to_R$. Esse conceito traz ferramentas necessárias para trabalhar mais genericamente propriedades dos Sistemas de Reescrita de Termos (TRS's) que trataremos na próxima seção.

Para simplificar o nosso trabalho, considere a seguinte notação:

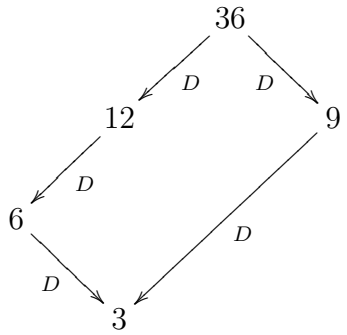
- ${}_R\!\leftarrow$ denota a relação inversa.
- $\to_R^=$ denota o fecho reflexivo da relação, ou seja, 1 ou 0 passos.
- $\to_R^+$ denota o fecho transitivo, ou seja, 1 ou mais passos da relação.
- $\to_R^*$ denota o fecho reflexivo transitivo, isto é, 0 ou mais passos da relação.
- ${}_R^=\!\leftarrow$, ${}_R^+\!\leftarrow$, ${}_R^*\!\leftarrow$ denotam, respectivamente, os fechos reflexivo, transitivo e reflexivo transitivo da relação inversa.

Quando o contexto for óbvio, usaremos apenas $\to$ ao invés de $\to_R$.

Considere, por exemplo, o sistema $(\mathbb{N}, \rightarrow_D)$, onde

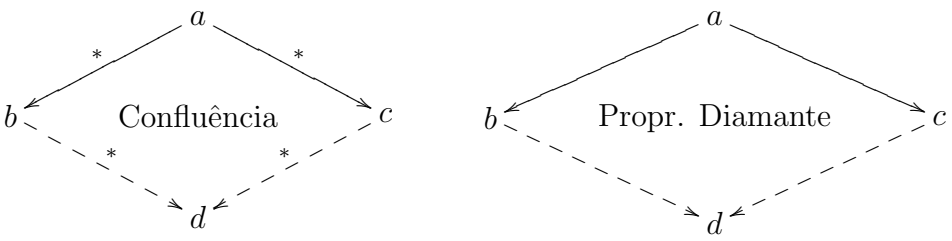
$$a \rightarrow_D b \text{ se, e somente se, } b|a \; \forall a, b \in \mathbb{N}.$$

Assim, temos que $36 \rightarrow_D 12 \rightarrow_D 6 \rightarrow_D \ldots$. No entanto, também sabemos que $36 \rightarrow_D 9 \rightarrow_D \ldots$. Assim, observamos uma divergência $6 \; {}_D\!\leftarrow 12 \; {}_D\!\leftarrow 36 \rightarrow_D 9$. Porém, podemos encontrar um elemento que junta 6 e 9 uma vez que $6 \rightarrow_D 3 \; {}_D\!\leftarrow 9$. Observe o diagrama:



Num ARS $(A, \rightarrow)$, para quaisquer $b, c \in A$, b e c são **juntáveis** sempre que existe $d \in A$ tal que $b \rightarrow^* d \; {}^*\!\leftarrow c$. Assim, $(A, \rightarrow)$ é um sistema **confluente** sempre que $\forall a, b, c \in A$, $b \; {}^*\!\leftarrow a \rightarrow^* c$ implica em b e c juntáveis.

Outra definição importante é a de **propriedade diamante**. Um sistema ter tal propriedade significa que qualquer divergência em um passo é juntável em um passo, isto é, num sistema $(A, \rightarrow)$, $\forall a, b, c \in A$, se $b \leftarrow a \rightarrow c$, então existe $d \in A$ tal que $b \rightarrow d \leftarrow c$.



Logo, o sistema $(\mathbb{N}, \rightarrow_D)$ é confluente e tem a propriedade diamante, pois para qualquer divergência, o elemento 1 é um elemento de juntabilidade uma vez que divide qualquer número natural.

1.2.2 Sistema de Reescrita de Termos

Termos são estruturas definidas indutivamente a partir de um conjunto de constantes e funções Σ e outro conjunto infinito enumerável de variáveis V, de modo que $\Sigma \cap V = \emptyset$ e:

1. variáveis são termos;

2. constantes são termos;

3. $f(t_1, t_2, \ldots, t_n)$ é um termo sempre que $f \notin V$ é uma função n-ária e, $\forall i = 1, \ldots, n$, temos que t_i é um termo.

O conjunto de termos formados a partir de Σ e de V é denotado por $T(\Sigma, V)$. De fato, as constantes podem ser vistas como funções com aridade 0. O **subtermo** t de um termo s em uma posição p pode ser escrito como $t = s|_p$ e denotamos por $s[p \leftarrow t_1]$ o termo resultante da troca do subtermo da posiçao p em s por t_1, ou seja, é o termo s a menos da posição p, onde fizemos a troca.

As **posições** de um termo t também são descritas indutivamente como sequências de naturais, sendo que:

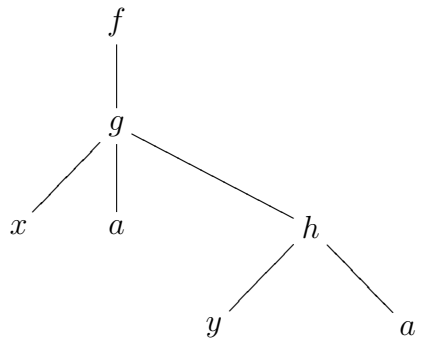
1. ϵ é a posição raiz e qualquer termo a tem como posição;

2. se t é uma variável, então sua única posição é ϵ;

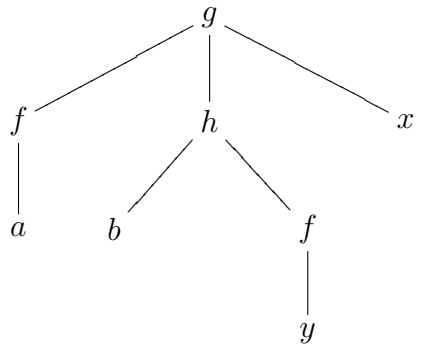
3. se $t = f(t_1, \ldots, t_n)$ é um termo funcional, então $\forall i = 1, \ldots, n$, i concatenado com uma posição de t_i é uma posição de t.

Veja alguns exemplos:

$t = f(g(x, a, h(y, a)))$ e $t|_{1.3} = h(y, a)$



$s = g(f(a), h(b, f(y)), x)$ e $s|_{2.2} = f(y)$



Também dizemos que duas posições p e q são **paralelas** sempre que não houver prefixo comum entre elas além da posição raiz. Por exemplo, as posições 2.3.1 e 2.3.5 não são paralelas, pois têm como prefixo comum a posição 2.3. Por outro lado, as posições 1.3.3 e 2.3 são paralelas pois não possuem prefixo comum, uma vez que o primeiro natural de uma posição é diferente do primeiro natural da outra.

Sistemas de Reescrita de Termos (TRS's) são programas com um número finito de **regras de reescrita**. Uma regra é basicamente um par de termos, que pode ser representado por $\langle l, r \rangle$ ou por $l \longrightarrow r$, sendo esta última a mais comum, de modo que o lado esquerdo l está relacionado com o lado direito r. Como tanto o lado esquerdo quanto

o lado direito de uma regra são termos, ambos devem assumir a estrutura apresentada para tais. No entanto, existem algumas restrições que devem ser observadas. O lado esquerdo l não deve ser uma variável e qualquer variável que ocorrer do lado direito r necessariamente deverá ocorrer em l também. Exemplo:

$$\begin{aligned} S: \quad & f(a) \longrightarrow g(b) \\ & h(x,y) \longrightarrow f(g(x)) \end{aligned}$$

Uma regra de reescrita é dita **linear à esquerda** quando o termo que ocorre ao lado esquerdo da referida regra possui no máximo uma ocorrência de determinada variável, ou seja, se uma variável aparecer, ela só ocorrerá uma vez. Um sistema é dito linear à esquerda se todas as suas regras são lineares à esquerda. O mesmo conceito de linearidade vale para o lado direito. Assim, um sistema será **linear** se for tanto linear à esquerda quanto linear à direita.

Linear à esquerda	Linear à direita	Linear
$S: \quad f(0,x) \longrightarrow f(x,g(x))$	$R: \quad 1 \longrightarrow 2$	$E: \quad g(x) \longrightarrow f(x,0)$
$\quad\quad h(y,x,1) \longrightarrow g(f(x,y))$	$\quad\quad h(x,x,y) \longrightarrow h(x,y,1)$	$\quad\quad h(1,x,y) \longrightarrow f(y,x)$
$\quad\quad g(1) \longrightarrow 0$	$\quad\quad g(f(y,y)) \longrightarrow f(a,b)$	$\quad\quad h(0,0,1) \longrightarrow g(1)$

Uma **substituição** σ é uma aplicação $\sigma : V \longrightarrow T(\Sigma, V)$, ou seja, leva variáveis em termos. Além disso, σ somente deixa de fixar um número finito de variáveis; assim, o domínio de σ, $Dom(\sigma)$, conjunto das variáveis que não são mapeadas nelas mesmas, é finito. A substituição σ pode ser extendida ao conjunto de termos; para $t\sigma$, por exemplo, basta trocar a ocorrência de cada variável do termo t pela sua respectiva imagem em σ. Uma substituição ρ é um **renomeamento** se a imagem de qualquer variável é uma variável.

Dois termos t e s são ditos **unificáveis** se existe uma substituição γ tal que $t\gamma = s\gamma$. Nesse caso, γ é chamado um **unificador** de t e s.

Em TRS's, o conceito de reescrita toma uma outra dimensão. Acrescentaremos alguns detalhes que tornarão o processo mais preciso em relação ao modo como podemos trabalhar, porém todos os outros conceitos que vimos e resultados que obteremos são ainda aplicáveis aqui. Dizemos que, num sistema E de regras, para $t, s \in T(\Sigma, V)$, t **reescreve** ou **reduz** para s (denotamos $t \longrightarrow_E s$ ou $t \longrightarrow s$ quando o sistema for óbvio no contexto) sempre que existir uma regra $e = \langle l, r \rangle \in E$ tal que, em alguma posição p de t, $t|_p = l\sigma$ e $s = t[p \leftarrow r\sigma]$.

$$\begin{aligned} E: \quad & h(x,y) \longrightarrow g(x,x) \\ & g(x, f(y)) \longrightarrow h(y,y) \end{aligned}$$

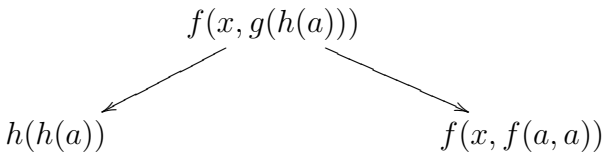
A partir do sistema E, é possível proceder com a sequência de reduções $h(f(a), h(f(b), b)) \longrightarrow_E h(f(a), g(f(b), f(b))) \longrightarrow_E h(f(a), h(b, b)) \longrightarrow_E g(f(a), f(a)) \longrightarrow_E h(a, a) \longrightarrow_E g(a, a)$. De fato, observemos a primeira redução. O subtermo de $t = h(f(a), h(f(b), b))$ na posição 2 é exatamente o lado esquerdo da primeira regra $l_1 = h(x, y)$ instanciado com a substituição $\sigma = \{x/f(b), y/b\}$ (leva x em $f(b)$ e y em b), isto é, $t|_2 = l_1\sigma$. Além disso, o termo resultante $s = h(f(a), g(f(b), f(b)))$ é o termo t, trocando o subtermo da posição 2 pelo lado direito da primeira regra $r_1 = g(x, x)$ instanciado também por σ, ou seja, $s = t[2 \leftarrow r_1\sigma]$.

Outro conceito importante é o de **par crítico**. Para que dois termos t e s sejam um par crítico, é necessária a existência de duas regras de reescrita $e_1 = \langle l_1, r_1\rangle$ e $e_2 = \langle l_2, r_2\rangle$ tais que os conjuntos de variáveis dos termos l_1 e l_2 sejam disjuntos (se não forem, basta renomear de forma que sejam), o subtermo de l_1 numa posição p, $l_1|_p$, não seja uma variável e $l_1|_p$ e l_2 sejam unificáveis com um unificador γ. Daí, devemos ter que $t = r_1\gamma$ e $s = l_1\gamma[p \leftarrow r_2\gamma]$ é o par crítico considerado. Em outras palavras, para que haja um par crítico, é necessário que o lado esquerdo de uma regra seja unificável com o subtermo do lado esquerdo de outra regra numa posição que não seja de variável. O par crítico é o produto da divergência dessas duas regras que se sobrepõem. Considere, por exemplo, o seguinte sistema:

$$A: \quad \begin{array}{l} f(x, g(y)) \longrightarrow h(y) \\ g(h(a)) \longrightarrow f(a, a) \end{array}$$

$\gamma = \{y/h(a)\}$ é um unificador para $g(y)$ e $g(h(a))$, pois $(g(y))\gamma = g(h(a))$ e γ não altera $g(h(a))$, uma vez que não há variáveis neste termo.

Assim, temos um par crítico, que é basicamente uma divergência a partir das regras onde ocorreu *matching*.



Quando ocorre um par crítico, nós dizemos que o sistema de regras é **ambíguo**. Na verdade, o que nos interessa principalmente são os sistemas não ambíguos, onde não há par crítico, e que sejam lineares à esquerda. Tais sistemas são chamados **ortogonais**.

1.2.3 Principais Teoremas

Nesta seção, veremos alguns resultados fundamentais até desenvolvermos a prova de que TRS's ortogonais são confluentes. Em alguns teoremas, veremos a ideia principal para a prova, sem apresentar todos os detalhes técnicos.

Teorema 1.2.1 *Todo sistema de reescrita que tem a propriedade do diamante é confluente.*

Demonstração: Seja o ARS $(A, \rightarrow)$ e $a, b, c \in A$ tais que $b \ ^*\!\leftarrow a \rightarrow^* c$. Precisamos provar que existe $d \in A$ tal que $b \rightarrow^* d \ ^*\!\leftarrow c$. Faremos a prova por indução sobre o número de passos m em que a atinge b e sobre o número de passos n em que a atinge c ($b \ ^m\!\leftarrow a \rightarrow^n c$).

Base de indução (BI) sobre m: se $m = 0$, então não há divergência, pois $a \rightarrow^0 b$ implica que $a = b$. Assim, já sabemos que $b = a \rightarrow^* c \ ^0\!\leftarrow c$. Logo, c é o nosso elemento de juntabilidade.

Passo indutivo (PI) sobre m: se $m \geq 1$, então sabemos que existe $b' \in A$ tal que $a \rightarrow b' \rightarrow^{m-1} b$.

BI sobre n: se $n = 0$, então não há divergência e b é o termo de juntabilidade, pois temos que $b \rightarrow^0 b \ ^*\!\leftarrow c = a$.

PI sobre n: se $n \geq 1$, então sabemos que existe $c' \in A$ tal que $a \rightarrow c' \rightarrow^{n-1} c$. Como temos a divergência em um passo $b' \leftarrow a \rightarrow c'$, pela hipótese da propriedade do diamante, existe $e \in A$ tal que $b' \rightarrow e \leftarrow c'$. Agora, temos a divergência $b \ ^{m-1}\!\leftarrow b' \rightarrow e$ e, pela hipótese de indução (HI) sobre m, existe $d' \in A$ tal que $b \rightarrow^* d' \ ^*\!\leftarrow e$. Finalmente, temos a divergência $d' \ ^*\!\leftarrow e \leftarrow c' \rightarrow^{n-1} c$. Pela HI sobre n, existe $d \in A$ tal que $d' \rightarrow^* d \ ^*\!\leftarrow c$. Pela transitividade do fecho reflexivo transitivo, $b \rightarrow^* d' \rightarrow^* d$ implica $b \rightarrow^* d \ ^*\!\leftarrow c$, como queríamos demonstrar. ■

A estratégia de prova utilizada em [3, 4] para a prova de que TRS's ortogonais são confluentes utiliza o fato de que uma segunda relação entre termos chamada **redução ou reescrita em paralelo** tem a propriedade diamante. Daí, pelo teorema anterior, temos a confluência para a redução em paralelo, que equivale à confluência para a reescrita simples, uma vez que o fecho reflexivo transitivo da primeira corresponde ao fecho reflexivo transitivo da segunda.

Nós definimos a reescrita em paralelo de modo semelhante ao da reescrita simples, sendo que a redução pode ocorrer em várias posições paralelas dentro do termo original. Denotaremos a reescrita em paralelo sobre as regras de um sistema E por $\rightrightarrows_E$. Assim, quando dizemos que $t_1 \rightrightarrows_E t_2$, estamos afirmando que há uma sequência (possivelmente vazia) $\Pi = (\pi_0, \ldots, \pi_n)$ de posições paralelas do termo t_1, uma sequência $\epsilon = (\langle l_0, r_0\rangle, \ldots, \langle l_n, r_n\rangle)$ de regras em E e uma sequência $\Gamma = (\gamma_0, \ldots, \gamma_n)$ de substituições, tais que, $\forall i = 0, \ldots, n$ $(n = |\Pi| - 1)$, temos:

1. $t_1|_{\pi_i} = l_i\gamma_i$;
2. $t_2|_{\pi_i} = r_i\gamma_i$ e as diferenças entre t_1 e t_2 ocorrem somente nessas posições.

Teorema 1.2.2 *$t_1 \rightrightarrows_E^* t_2$ se, e somente se, $t_1 \to_E^* t_2$.*

Demonstração:

($\Rightarrow$) De fato, como para cada passo de $\rightrightarrows_E$ nós temos vários passos de $\to_E$, então $t_1 \rightrightarrows_E^* t_2$ corresponde a $t_1(\to_E^*)^* t_2$. Mas isso corresponde a $t_1 \to_E^* t_2$ e temos o que queríamos.

($\Leftarrow$) Nós podemos considerar que cada passo de $\to_E$ corresponde a um passo de $\rightrightarrows_E$. É só tomar a sequência unitária de posição, de regra e de substituição. Assim, já temos diretamente que $t_1 \to_E^* t_2$ implica que $t_1 \rightrightarrows_E^* t_2$. ■

Para provar o teorema seguinte, consideraremos verdadeiro que a relação $\rightrightarrows_E$ tem a propriedade diamante no caso em que E seja ortogonal e, posteriormente, faremos um esboço da demonstração da afirmação.

Teorema 1.2.3 *Seja E um sistema de regras de reescrita ortogonal. Então, a relação $\to_E$ é confluente.*

Demonstração: Assumiremos que $\rightrightarrows_E$ tem a propriedade diamante. Pelo teorema 1.2.1, sabemos então que $\rightrightarrows_E$ é confluente, o que quer dizer que, $\forall s, t_1, t_2 \in T(\Sigma, V)$ tais que $t_1 \,{}^*\!\!\leftleftarrows t \rightrightarrows^* t_2$ implica que $\exists u \in T(\Sigma, V)$ tal que $t_1 \rightrightarrows^* u \,{}^*\!\!\leftleftarrows t_2$.

Como temos a equivalência entre $\to_E^*$ e $\rightrightarrows_E^*$, segue que $\to_E$ é confluente. ■

Teorema 1.2.4 *Seja E um sistema de regras de reescrita ortogonal. Então, a relação $\rightrightarrows_E$ tem a propriedade diamante.*

Esboço da prova: Sejam $s, t_1, t_2 \in T(\Sigma, V)$ tais que $t_1 \leftleftarrows s \rightrightarrows t_2$. Então, existem duas sequências Π_1 e Π_2 de posições paralelas de s onde ocorrem as reduções. No entanto, essas posições podem não ser paralelas quando se considera as posições das duas sequências juntas.

Consideraremos as posições onde ocorrerão as reduções como sendo o vértice superior dos triângulos coloridos na Figura 1.1. Os triângulos vermelhos e o roxo representam os subtermos de s que serão modificados para atingir o termo t_1 em um único passo de reescrita em paralelo; os triângulos azuis e o roxo representam os subtermos de s que serão modificados para atingir o termo t_2.

Como esses subtermos serão reescritos, isso significa que fazem *matching* com alguma instanciação do lado esquerdo de regras que estão no sistema ortogonal E. Chamaremos de ϵ_1 e Γ_1 as sequências de regras e de substituições, respectivamente, responsáveis pela

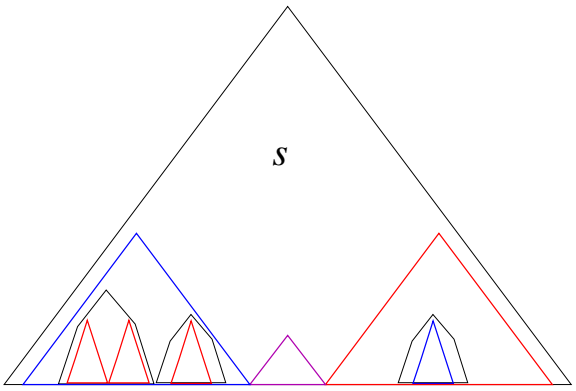


Figura 1.1: Termo inicial

reescrita para t_1 e de ϵ_2 e Γ_2 as sequências de regras e substituições, respectivamente, responsáveis pela reescrita para t_2.

Assim, como a Figura 1.1 mostra, de fato há a possibilidade de encontrarmos o lado esquerdo de regras em ϵ_1 instanciado com as substituições de Γ_1 dentro do lado esquerdo de regras em ϵ_2 instanciado com as substituições em Γ_2 e vice-versa. Note que há pentágonos interpostos entre os triângulos sempre que há "sobreposição" de regras. De fato, não há sobreposição, pois esses pentágonos representam variáveis cuja instanciação pelas substituições correspondentes ao termo colorido maior contém os subtermos coloridos. A existência dessas variáveis é devida à não ambiguidade das regras em E. No caso do subtermo roxo, como as posições coincidem, também pela não ambiguidade já sabemos que é uma única regra que faz a redução para o termo t_1 e para t_2.

Daí, temos a seguinte divergência quando acontecem as reescritas em paralelo:

Os triângulos preenchidos representam o lado direito das regras, ou seja, onde já ocorreram as modificações, mas ainda assim pode haver triângulos vazados no seu interior. Isso acontece porque as variáveis que contêm o lado esquerdo de algumas regras podem ocorrer também do lado direito de uma regra; mais que isso, as variáveis podem se replicar no lado direito. De fato, quando nós falamos de ortogonalidade, falamos de linearidade à esquerda, mas à direita não há restrições quanto ao número de ocorrências de uma dada variável.

Note ainda que as posições das variáveis podem ser modificadas dentro do termo colorido maior. Assim, os subtermos coloridos das variáveis também assumirão novas posições, podendo se replicar.

O nosso objetivo é obter um termo u que satisfaça o que acontece na figura seguinte:

Para tanto, vamos analisar o que deve ocorrer para que $t_1 \rightrightarrows_E u$. Entendendo como isso ocorre, sabemos que $t_2 \rightrightarrows_E u$ é um caso simétrico a este.

Como falamos anteriormente, o termo de contorno roxo não será modificado de t_1 para u, pois não há diferença entre t_1 e t_2 nessa posição.

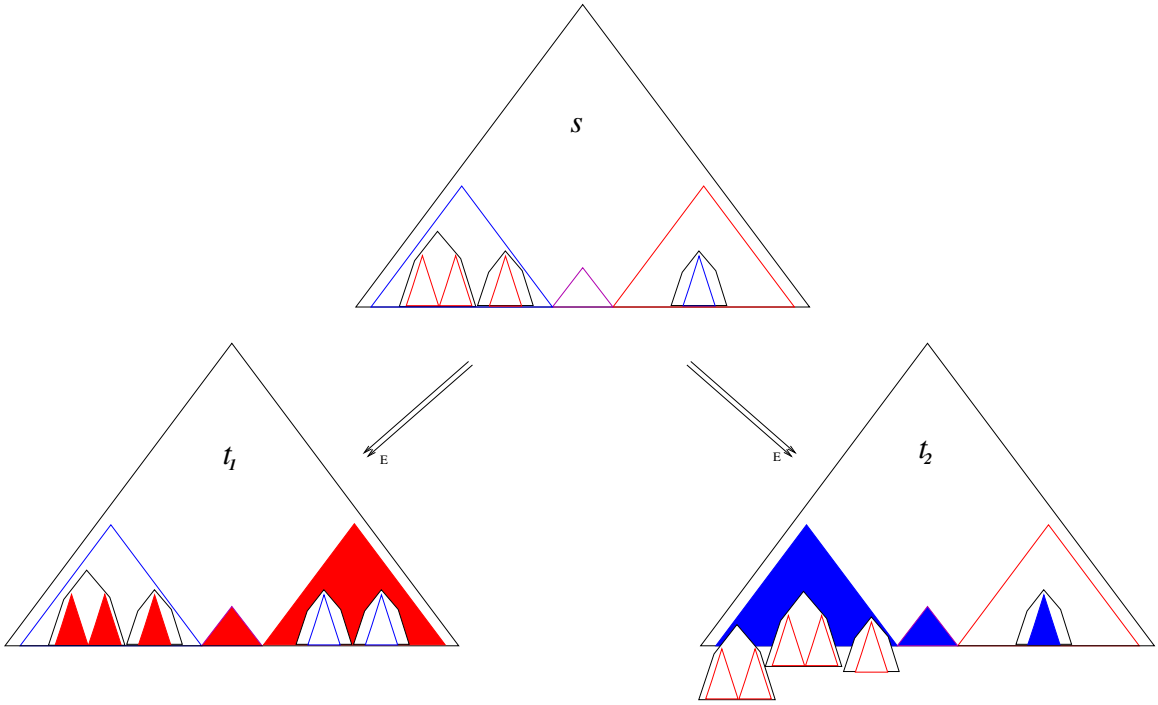


Figura 1.2: Divergência em um passo de reescrita em paralelo

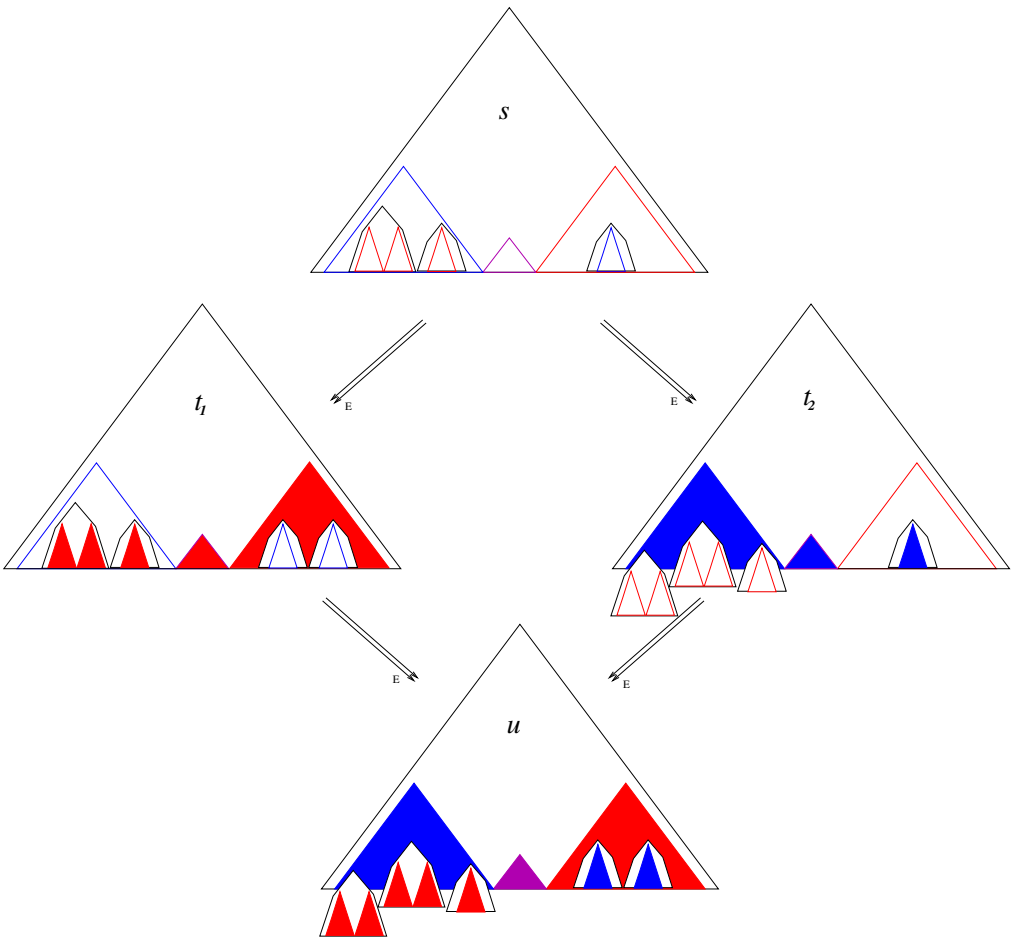


Figura 1.3: Termo de juntabilidade

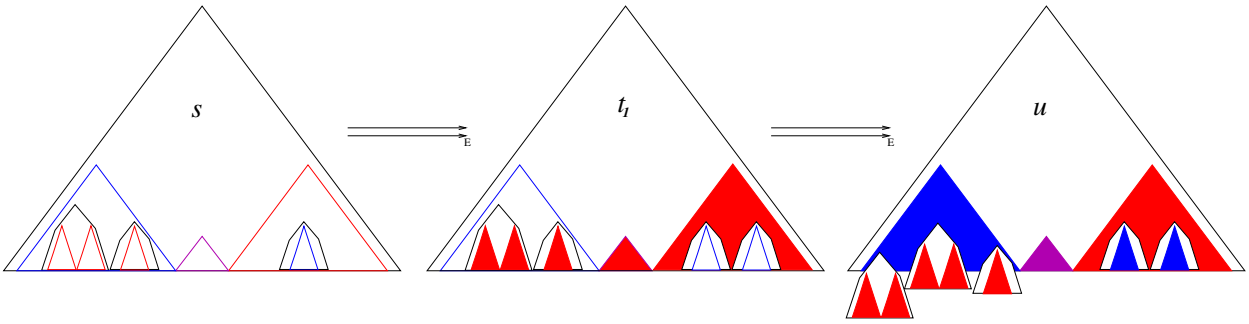


Figura 1.4: Cadeia de reduções passando por t_1

Os subtermos vazados azuis que estão dentro do triângulo vermelho preenchido precisam ficar preenchidos também. A única observação ao se reduzir esses termos é que precisamos atualizar as posições, uma vez que as variáveis que os contêm trocam as posições no triângulo preenchido, isto é, quando o triângulo vermelho deixa de ser lado esquerdo para ser lado direito da regra instanciada. Nesse caso, as regras que utilizaremos são as de ϵ_2 e as substituições de Γ_2 que as acompanham.

Em relação ao triângulo azul vazado grande em t_1, há que se tomar cuidado. Observe que a substituição que instanciava essa regra em s não serve mais, pois o termo foi modificado em suas "subposições". No entanto, a mesma substituição pode ser alterada de forma que a mesma regra faça *matching* com esse termo em t_1. Nesse momento, é importante lembrar que as substituições agem somente sobre variáveis. Mas as alterações que o termo do triângulo azul grande sofreu ocorreram justamente dentro das instanciações de variáveis, ou seja, dentro dos pentágonos. Assim, criamos uma nova substituição que age exatamente como a de antes, mas onde aparecem as nossas variáveis de interesse, trocam-se os triângulos vermelhos vazados (como em s) pelos triângulos vermelhos preenchidos (como em t_1).

Logo, chegamos ao fim do nosso esboço pois encontramos as posições, as regras e as substituições que precisávamos encontrar para fazer a redução em paralelo de t_1 para u. O outro caso, é análogo. ■

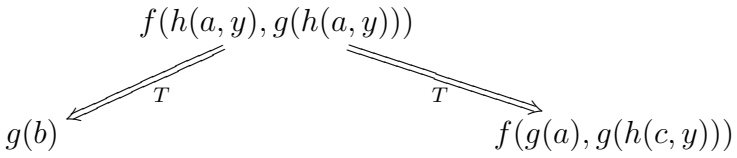
A mesma ideia apresentada aqui será utilizada para a formalização do teorma em um assistente de prova: primeiramente descrevemos quem é o termo de juntabilidade e, depois, nos damos conta de que existem sequências de posições, regras e substituições que satisfazem as condições para redução em paralelo.

A não ambiguidade é fundamental nessa prova uma vez que é dela que sacamos a existência das variáveis onde podemos agir para criar novas substituições. Por outro lado, a linearidade à esquerda é importante pois a reescrita em paralelo age em posições específicas enquanto que a substituição age em todas as ocorrências das variáveis em seu domínio. Isso acarretaria diferenças no termo final. Para entendermos essa diferença, acompanhemos o exemplo que temos a seguir.

Considere o sistema T não ambíguo, porém não linear à esquerda.

$$\begin{aligned} T: \quad & f(x, g(x)) \longrightarrow g(b) \\ & h(a, x) \longrightarrow g(a) \\ & a \longrightarrow c \end{aligned}$$

Acompanhe a divergência que ocorre a partir do termo $s = f(h(a, y), g(h(a, y)))$.

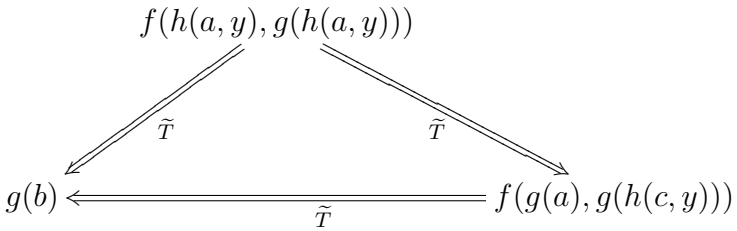


Assim, não existe substituição capaz de instanciar a primeira regra de tal forma que ocorra o *matching* do lado esquerdo com o termo $f(g(a), g(h(c, y)))$, pois a imagem de x teria que ser $g(a)$ e $h(c, y)$ ao mesmo tempo.

Agora, vamos fazer uma pequena modificação em T e criar o sistema ortogonal $\widetilde{T}$.

$$\begin{aligned} \widetilde{T}: \quad & f(x, g(y)) \longrightarrow g(b) \\ & h(a, x) \longrightarrow g(a) \\ & a \longrightarrow c \end{aligned}$$

Novamente, nós temos a mesma divergência a partir do termo s.



No entanto, a divergência é juntável. Quando fizemos $f(h(a, y), g(h(a, y))) \rightrightarrows_{\widetilde{T}} g(b)$, utilizamos a primeira regra e a substituição $\sigma = \{x/h(a, y)\,,\, y/h(a, y)\}$. Finalmente, para fazer $f(g(a), g(h(c, y))) \rightrightarrows_{\widetilde{T}} g(b)$, utilizamos a mesma regra e a substituição $\sigma' = \{x/g(a), y/h(c, y)\}$.

Capítulo

2

Especificação

Toda a teoria apresentada aqui tem se especificado com o objetivo de formalizá-la em um assistente de prova semi-automático. O assistente utilizado no trabalho é o *Prototype Verification System* (PVS) na versão 5.0, provador baseado numa lógica de ordem superior.

O nome da teoria desenvolvida é `orthogonality` e utiliza grande parte das teorias `ars` e `trs` desenvolvidas pelo Grupo de Teoria da Computação (GTC/UnB), disponíveis em [1], na forma dos próprios arquivos de formalização, e em [2, 6, 7], como artigos apresentados pelos desenvolvedores do GTC/UnB.

2.1 Sistemas Abstratos de Reescrita

Algumas definições tratadas nesses trabalhos nos darão base para entender a especificação desenvolvida aqui. Apresentamos primeiramente as que se relacionam aos ARS's. Esses conceitos estão disponíveis na teoria `ars` como mencionado acima.

```
RC(R): reflexive = union(R, =)

iterate(f, n)(x): RECURSIVE T =
  IF n = 0 THEN x ELSE f(iterate(f, n-1)(x)) ENDIF
  MEASURE n

TC(R): transitive = IUnion(LAMBDA (p:posnat): iterate(R, p))

RTC(R): reflexive_transitive = IUnion(LAMBDA (n:nat): iterate(R, n))

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

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

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

Note que `R` é uma relação de reescrita. Até este momento, ainda estamos trabalhando somente as definições relacionadas aos ARS's. Temos que `RC` (*reflexive closure*) nos dá o fecho reflexivo da relação `R`, isto é, se temos `RC(R)(x,y)`, então `R(x,y)` ou `x=y`. Já a função `iterate` faz a função `f` agir sobre `x` por `n` vezes, sendo definida recursivamente, nos resultando numa multi-composição da mesma função. O fecho transitivo da relação `R` é denotado por `TC(R)` e é definido como a união de todas as composições de `R` por um número `p` natural positivo de vezes (`p`$\geq$ 1). O fecho reflexivo transitivo de `R`, denotado por `RTC(R)`, é definido da mesma forma, mas para `n` natural qualquer, ou seja, há a possibilidade de que não haja iteração alguma de `R` se `n=0`.

Já a noção de juntabilidade de dois elementos `x` e `y` através da relação `R` é dada pelo predicado `joinable?`, onde `joinable?(R)(x,y)` significa que há um elemento `z` tal que `x` e `y` atingem `z` através de `RTC(R)`, ou seja, existe uma cadeia de redução de `x` a `z` com 0 ou mais passos e outra cadeia de redução também em 0 ou mais passos de `y` a `z`. O predicado `confluent?` aponta que uma determinada relação `R` é confluente, isto é, para qualquer divergência a partir de um elemento `x`, `RTC(R)(x,y)` e `RTC(R)(x,z)`, esses elementos `y` e `z` são juntáveis. A propriedade diamante é expressa pelo predicado `diamond_property?` que avalia se uma relação `R` tem ou não tal propriedade o que quer dizer que, se `diamond_property?(R)` é verdadeira, qualquer divergência em um passo para cada lado (`R(x,y)` e `R(x,z)`) implica na existência de um elemento de juntabilidade `r` que pode ser alcançado em um passo de cada lado (`R(y,r)` e `R(z,r)`).

2.2 Sistemas de Reescrita de Termos

Outras definições chave para a compreensão do código são os conceitos relacionados aos TRS's, tratados anteriormente ao presente trabalho e disponíveis na teoria `trs`. São estes:

```
term: TYPE

vars?, app?: [term -> boolean]

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

vars: [variable -> (vars?)]

app:
      [[f: symbol,
```

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

v: [(vars?) -> variable]

f: [(app?) -> symbol]

args:
      [d: (app?) ->
         {args: finite_sequence[term] | args`length = arity(f(d))}]

term_v_vars: AXIOM
  FORALL (vars1_var: variable): v(vars(vars1_var)) = vars1_var;

term_f_app: AXIOM
  FORALL (app1_var: symbol,
          app2_var:
            {args: finite_sequence[term] |
                     args`length = arity(app1_var)}):
    f(app(app1_var, app2_var)) = app1_var;

term_args_app: AXIOM
  FORALL (app1_var: symbol,
          app2_var:
            {args: finite_sequence[term] |
                     args`length = arity(app1_var)}):
    args(app(app1_var, app2_var)) = app2_var

term_inclusive: AXIOM
  FORALL (term_var: term): vars?(term_var) OR app?(term_var)
```

Foi criado o tipo `term` cujos elementos podem ser avaliados pelos predicados `vars?` (que julga se um dado termo é uma variável) e `app?` (diz se é um termo funcional). `V` é definido como o conjunto de todas as variáveis. A função `vars` toma qualquer variável e lhe dá o valor de verdade como variável pelo predicado `vars?` e a função `app` faz o correspondente em relação às aplicações, termos funcionais com um símbolo `f` e uma sequência de termos `args` aplicados como argumentos em cada posição de `f` cujo comprimento coincide com a aridade do símbolo.

As funções `v`, `f` e `args` fazem o oposto, tomando o que tem valor de verdade como variável no caso de `v` ou aplicação no caso de `f`/`args`, retornando a variável avaliada por `vars?`, ou ainda, para o termo avaliado como funcional por `app?`, o símbolo de função ou os argumentos do termo, respectivamente. Vale mencionar que essa parte é gerada automaticamente pelo PVS quando definimos um *datatype*. Os axiomas `term_v_vars`,

`term_f_app` e `term_args_app` mostram justamente essas propriedades descritas acima. O axioma `term_inclusive` restringe as opções de caracterização de um termo, que só pode ser uma variável ou uma aplicação.

2.3 Posições

```
position: TYPE = finseq[posnat]

positions: TYPE = set[position]

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 <<
```

Como tratamos antes, cada posição, denotada por `position`, é uma sequência finita de naturais não nulos (`finseq[posnat]`) e `positions` é um conjunto de posições (`set[position]`). O conjunto de posições de um termo `t`, denotado por `positionsOF(t)`, é definido recursivamente onde, se `t` é uma variável, então o conjunto será `only_empty_seq`, ou seja, a única posição de `t` é `empty_seq`, a posição raiz; senão, `t` é uma aplicação `app(f,st)` e, caso a sequência de argumentos `st` aplicados a `f` tenha comprimento nulo, então temos novamente `only_empty_seq` e, caso contrário, temos que `positionsOF(t)` será a união de `only_empty_seq` e as posições formadas pela ação de concatenar cada índice `i` da sequência `st` adicionando 1, isto é, `i+1`, com as posições do termo dentro do argumento na posição `i` da sequência (é somado 1 ao índice, pois trabalhamos com `posnat` em relação às posições de termos, enquanto que utilizamos `nat` nos índices de uma sequência finita).

```
      p, q: VAR position
       fsp: VAR finseq[position]
         t: VAR term
```

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

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

PP?(fsp): bool = IF fsp`length < 2
                 THEN true
                 ELSE
                   FORALL (i, j: below[length(fsp)]): i /= j  =>
                                  parallel(fsp(i), fsp(j))
                 ENDIF

SP?(t)(fsp): bool = FORALL (i: below[length(fsp)]):
                       positionsOF(t)(fsp(i))

SPP?(t)(fsp): bool = PP?(fsp) & SP?(t)(fsp)
```

Temos que a posição `p` é prefixo de `q` sempre que existir uma posição complemento `p1`, ou seja, `q` é `p` concatenada com `p1`. Por exemplo, se `p`= 3.2 e `q`= 3.2.2.4.5, então temos que `p<=q`, pois `q=p o p1`, onde `p1`= 2.4.5. Agora, vale `parallel(p,q)` se não vale a relação binária `<=` em nenhum dos dois sentidos. Sendo assim, algumas propriedades são extraídas facilmente como, por exemplo, a comutatividade de `parallel`, que vem diretamente da comutatividade do conectivo lógico `&`. O predicado `PP?` aplicado a uma sequência de posições `fsp` compara todas as posições da sequência e verifica se todas são paralelas duas a duas. Já `SP?(t)(fsp)` é verdade se todas as posições de `fsp` são posições do termo `t` e vale `SPP?(t)(fsp)` se valem as duas proposições acima (`PP?(fsp)` e `SP?(t)(fsp)`).

2.4 Sequências finitas

Antes de continuar, vale a pena trazer alguns conceitos sobre sequências finitas (`finseq`'s) que darão base a várias definições daqui para frente. É importante dizer que a teoria que trata esses conceitos `finite_sequences_extras[T: TYPE]` trabalha com um tipo arbitrário `T`, sendo que a teoria pode ser importada e instanciada tranquilamente por qualquer outro tipo existente, o que faremos extensamente durante a especificação. Esse modelo de tipagem é chamado polimorfismo.

```
first(seq: not_empty_seq): T = seq(0)

rest(seq): finseq = IF seq`length = 0
                    THEN seq
                    ELSE ^(seq,(1, seq`length - 1))
```

```
                    ENDIF

delete(seq, (n: below[length(seq)])): finseq =
  (IF seq`length = 0
   THEN seq
   ELSE (# length := seq`length - 1,
           seq := (LAMBDA (i: below[seq`length - 1]):
                          (IF i < n THEN seq(i)
                           ELSE seq(i + 1)
                           ENDIF)) #)
   ENDIF)

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

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

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

A função `first` pode ser aplicada a uma `finseq` não vazia `seq` e retorna o primeiro elemento da sequência `seq(0)` de tipo `T`. Por outro lado, `rest` pode ser aplicada a qualquer sequência finita e retorna outra `finseq`, que é o resto da sequência (retira o primeiro elemento, se houver). A função `delete` que tem como argumentos uma sequência `seq` e um natural `n` menor que o comprimento da sequência, retorna `seq` a menos do elemento `seq(n)`, que é retirado, e os índices posteriores a `n` são ajustados. Quando `insert?` é aplicado a um elemento `x` de tipo `T`, a `seq`, que é uma `finseq[T]`, e a `n` natural menor ou igual ao comprimento de `seq`, temos que é acrescentado a `seq` o elemento `x` na posição de índice `n`, ajustando-se os índices posteriores. A aplicação `add_first` utiliza `insert?` para inserir o elemento `x` na primeira posição da sequência `seq`, ou seja, a de índice 0. Por último, temos a função `replace` que toma em `seq` o elemento `seq(n)`, que está na posição `n+1`-ésima, e o troca por `x`.

2.5 Subtermos

Voltamos a tratar dos conceitos dentro dos TRS's. Repare a ocorrência das notações apresentadas para `finseq[T]`, sendo que temos agora o tipo `T` instanciado com `position` e `term`, por exemplo, e vários outros tipos que precisaremos mais à frente.

```
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)

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

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

A aplicação `subtermOF` nos dá o subtermo de `t` na posição `p`, onde `p` pertence ao conjunto de posições de `t`. O algoritmo trabalha recursivamente através do comprimento de `p`. Enquanto o comprimento da posição que nós trabalhamos não for 0, então `t` é um termo funcional e nós podemos tomar o argumento `i`-ésimo (de índice `i-1` na sequência), onde `i` é o primeiro `posnat` na formação de `p`, e nós aplicamos novamente `subtermOF` a esse argumento e à posição `rest(p)`. Assim, o comprimento da posição sempre vai diminuindo, uma vez que aplicamos `rest` à posição anterior em cada passo. Quando obtivermos `p=empty_seq`, isto é, `length(p)=0`, então podemos parar e retornar o próprio termo aplicado naquele passo. O conjunto `Vars(t)` engloba todas as variáveis que ocorrem em `t` baseado na existência de posições desse termo cujos subtermos sejam variáveis. Já o conjunto `Pos_var(t,x)` abarca as posições de `t` onde ocorre uma dada variável `x`.

2.6 Substituições

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

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

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

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

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 <<
```

Declaramos `sigma` como uma aplicação que leva variáveis em termos e definimos o domínio de `sigma`, `Dom(sigma)`, como o conjunto das variáveis cuja imagem por `sigma` não são elas mesmas. Já `Ran(sig)` é o conjunto imagem de `sigma` sobre `Dom(sigma)` exclusivamente. Nós dizemos que `sigma` é uma substituição (denotado por `Sub?(sigma)`) sempre que `Dom(sigma)` é finito. Nós também estendemos a aplicação `sigma` a qualquer termo `t` homomorficamente, ou seja, fazemos `ext(sigma)(t)`, sendo que, se `t` for funcional, então aplicamos a extensão da substituição novamente a cada argumento do termo.

2.7 Redução simples

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

```
    ENDIF)
  MEASURE length(p)
```

O algoritmo `replaceTerm(s,t,p)` toma o termo `s` e troca o subtermo da posição `p` pelo termo `t`. Como em `subtermOF`, `replaceTerm` trabalha sobre o comprimento de `p`. Se `p` for a posição raiz, então basta trocar `s` por `t`. Senão, `s` é funcional. Então, nós atualizamos um dos argumentos que são aplicados ao símbolo de `s` e continuamos trabalhando com o `rest(p)`. A troca feita na sequência de argumentos em cada passo é realizada por `replace`.

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

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

Temos também o predicado `rewrite_rule?` que avalia que um par de termos `(l,r)` é uma regra de reescrita sempre que `l` não for uma variável e `Vars(r)` estiver contido em `Vars(l)`. Enfim, nós dizemos que `reduction?(E)(s,t)` sempre que o termo `s` reduzir para `t` com as regras do TRS `E`, isto é, se houver uma posição `p` de `s`, uma substituição `sigma` e uma regra `e` em `E` tais que o subtermo de `s` em `p` é `ext(sigma)(lhs(e))` e o termo resultante `t` é `replaceTerm(s, ext(sigma)(rhs(e)), p)`, onde `lhs` e `rhs` são o lado esquerdo e direito de uma regra respectivamente.

2.8 Ortogonalidade

Essas foram as principais definições utilizadas daquilo que já havia sido realizado em [6–8]. Passamos agora a introduzir o que foi produzido durante o presente trabalho. O nome da teoria criada para tratar os nossos conceitos e resultados é `orthogonality`. O teorema que motivou o desenvolvimento da teoria foi o seguinte:

```
Orthogonal_implies_confluent: LEMMA
 FORALL (E : Orthogonal) :
   LET RRE = reduction?(E) IN
   confluent?(RRE)
```

No entanto, ainda nos cabe mostrar como definimos o conceito de `Orthogonal` que aparece aqui.

```
Ambiguous?(E): bool =  EXISTS (t1, t2) : CP?(E)(t1,t2)

linear?(t): bool = FORALL (x | member(x,Vars(t))) : Card[position](Pos_var(t,x)) = 1

Left_Linear?(E): bool = FORALL (e | member(e, E)) : linear?(lhs(e))

Orthogonal?(E): bool =  Left_Linear?(E) & NOT Ambiguous?(E)

Orthogonal: TYPE = (Orthogonal?)
```

A noção de ambiguidade (`Ambiguous?`) sobre um sistema de reescrita `E` dita a existência de dois termos que formem um par crítico nesse sistema, como indicamos na seção 1.2. O predicado `linear?` avalia um termo `t` quanto à ocorrência de suas variáveis, tomando cada variável `x` em `Vars(t)` e conferindo se a cardinalidade de `Pos_var(t,x)` é 1. Assim, um sistema é dito `Left_Linear?` se o termo do lado esquerdo de cada regra `lhs(e)` for `linear?`. Finalmente, temos que um sistema é tido como `Orthogonal?` se for `Left_Linear?` e não for `Ambiguous?`. `Orthogonal` é o tipo de todo sistema que seja `Orthogonal?`.

2.9 Definições auxiliares

Durante a prova do teorema principal, nós precisamos criar algumas definições que nos auxiliarão. Elas são mais básicas e tratam sequências finitas de tipo `T` instanciável. Como estamos tratando de tipos polimórficos mais uma vez, como na subseção 2.4, precisamos criar subteorias para tratar conceitos que envolvem um tipo arbitrário `T`. Tal tipo poderá ser instanciado como termo, posição, substituição, regra de reescrita e outros, conforme a conveniência.

```
mem_seq(x, seq): RECURSIVE bool =
    IF length(seq) = 0 THEN false
        ELSIF x = seq(0) THEN true
           ELSE mem_seq(x, rest(seq))
    ENDIF
    MEASURE length(seq)

seq2set(seq):  RECURSIVE set[T] =
    IF seq`length = 0 THEN emptyset
        ELSE add(seq(0), seq2set(rest(seq)))
    ENDIF
    MEASURE length(seq)

power(seq,n):
```

```
RECURSIVE finseq[T] = IF n=0 THEN empty_seq
       ELSE seq o power(seq,n-1)
 ENDIF MEASURE n

index(seq,x): RECURSIVE nat =
    IF seq‘length=0 THEN 0
       ELSIF x = seq‘seq(0)
             THEN 0
             ELSE 1+index(rest(seq),x)
 ENDIF MEASURE(seq‘length)

choose_seq(seq:PP, seq1:PP, (seq2 | seq1‘length=seq2‘length)):
RECURSIVE finseq[T] =
    IF length(seq1)=0 THEN empty_seq
        ELSIF mem_seq(seq1(0),seq)
           THEN add_first(seq2(0),choose_seq(seq,rest(seq1),rest(seq2)))
           ELSE choose_seq(seq,rest(seq1),rest(seq2))
    ENDIF
    MEASURE(length(seq1))
```

Temos o predicado `mem_seq`, que é aplicado a um objeto `x` de tipo `T` e uma sequência `seq`, julgando se `x` ocorre em `seq`. A função `power` toma a sequência `seq` e a replica `n` vezes. Já o algoritmo `seq2set` toma a sequência finita `seq` e a leva no conjunto formado pelos seus argumentos. O recurso `index` é utilizado para extrair o primeiro índice da sequência `seq` onde ocorre o elemento `x`. Por outro lado, se `x` não ocorre em `seq`, então `index(seq,x)=length(seq)`, que não é um índice de `seq`.

A função `choose_seq` por sua vez, toma duas sequências finitas de posições paralelas `seq` e `seq1`, além de outra sequência `seq2` com elementos de tipo `T` que tenha o mesmo comprimento de `seq1`. Daí, `choose_seq` retorna como saída uma sequência cujos argumentos ocorrem em `seq2` com os mesmos índices dos argumentos de `seq1` que ocorrem em `seq`, ou seja, verifica-se se cada termo de `seq1` ocorre em `seq` e, quando isso acontecer, selecionamos o elemento de `seq2` com índice correspondente à posição em `seq1`. É interessante perceber que `choose_seq` preserva ordem e, se `seq` for uma subsequência de `seq1`, também preserva comprimento de `seq`.

2.10 Redução em paralelo

```
replace_par_pos(s,(fsp:SPP(s)),fse|fse‘length=fsp‘length,fss|fss‘length=fsp‘length):
  RECURSIVE term =
    IF length(fsp) = 0
       THEN s
```

```
        ELSE replace_par_pos(replaceTerm(s,ext(fss(0))(rhs(fse(0))),fsp(0)),
                             rest(fsp),rest(fse),rest(fss))
      ENDIF
    MEASURE length(fsp)

parallel_reduction?(E)(s,t): bool =
  EXISTS (fsp: SPP(s),fse|(FORALL(i:below[fse‘length]):member(fse‘seq(i),E)),fss):
    fsp‘length = fse‘length AND fsp‘length = fss‘length
    AND (FORALL (i:below[fsp‘length]):subtermOF(s,fsp(i))=ext(fss(i))(lhs(fse(i))))
    AND t=replace_par_pos(s,fsp,fse,fss)
```

Esta aplicação `replace_par_pos` faz algo parecido com `replaceTerm`, mas a ação se realiza em várias posições paralelas do mesmo termo e de uma forma mais específica. Nesse caso, temos uma sequência `fsp` de posições paralelas de `s`, uma de regras de reescrita `fse` e uma de substituições `fss`, sendo que as três têm o mesmo comprimento. Assim, `replace_par_pos` troca o subtermo de `s` em cada posição de `fsp` pelo termo que fica do lado direito de cada regra em `fse` instanciado com as substituições de `fss`. As trocas começam pelo primeiro argumento de cada uma das sequências e o passo recursivo continua trabalhando com o `rest`. Todo esse trabalho é feito para construir a redução em paralelo que vimos na seção 1.2.3. Aqui, ela se chamará `parallel_reduction?`. O sistema `E` e os termos relacionados `s` e `t` são dados, enquanto que `parallel_reduction?` deduz a existência das sequências de posições, de regras e de substituições tais que o lado esquerdo das regras ocorram em `s` nas posições correspondentes descritas pela sequência de posições sendo instanciado pelas respectivas substituições, e o termo `t` seja fruto de aplicarmos `replace_par_pos` a `s` com essas três sequências.

2.11 Comparadores de sequências de posições

Ainda que estejamos sempre tratando de sequências de posições paralelas, quando tomamos duas dessas sequências, não temos mais a garantia de que as posições permanecem paralelas. Daí, podemos particionar as posições de uma dada sequência em relação a outra de acordo com as seguintes aplicações:

```
sub_pos((fsp : PP), p):
  RECURSIVE finseq[position] =
    IF length(fsp) = 0 THEN empty_seq[position]
       ELSIF p <= fsp(0) AND p /= fsp(0)
                THEN add_first(fsp(0), sub_pos(rest(fsp), p))
                ELSE sub_pos(rest(fsp), p)
    ENDIF
    MEASURE length(fsp)
```

```
Pos_Over((fsp1 : PP), (fsp2 : PP)):
  RECURSIVE finseq[position] =
    IF length(fsp1) = 0
       THEN empty_seq
       ELSIF length(sub_pos(fsp2, fsp1(0))) > 0
               OR PP?(add_first(fsp1(0), fsp2))
             THEN add_first(fsp1(0), Pos_Over(rest(fsp1), fsp2))
             ELSE Pos_Over(rest(fsp1), fsp2)
    ENDIF
  MEASURE length(fsp1)

Pos_Under((fsp1 : PP), (fsp2 : PP)):
  RECURSIVE finseq[position] =
    IF length(fsp2)=0
       THEN empty_seq
       ELSE sub_pos(fsp1, fsp2(0)) o Pos_Under(fsp1,rest(fsp2))
    ENDIF
  MEASURE length(fsp2)

Pos_Equal((fsp1 : PP), (fsp2 : PP)):
  RECURSIVE finseq[position] =
    IF length(fsp1) = 0 THEN empty_seq
       ELSIF mem_seq(fsp1(0),fsp2)
             THEN add_first(fsp1(0),Pos_Equal(rest(fsp1),fsp2))
             ELSE Pos_Equal(rest(fsp1), fsp2)
    ENDIF
  MEASURE length(fsp1)
```

Cada um desses quatro construtores nos dão como produto sequências de posições paralelas. Para a função `sub_pos`, temos como argumentos uma posição `p` e uma sequência de posições paralelas `fsp`; para as outras três, temos duas sequências de posições paralelas `fsp1` e `fsp2` em cada uma. A aplicação `sub_pos` constrói uma sequência com as posições de `fsp` que estejam por baixo de `p`, ou seja, tais que `p` lhes seja prefixo sem coincidir. A partir dela, as funções `Pos_Over` e `Pos_Under` comparam as posições das sequências `fsp1` e `fsp2`. No caso de `Pos_Over`, seleciona-se as posições de `fsp1` tais que sejam paralelas a todas as posições de `fsp2` ou que a sequência formada pelo `sub_pos` dessas posições em `fsp1` sobre `fsp2` não seja vazia, isto é, as que estejam por cima de alguma posição de `fsp2`. Por outro lado, `Pos_Under` seleciona as posições de `fsp1` que estejam por baixo de cada posição de `fsp2`. Por fim, temos a aplicação `Pos_Equal` que seleciona as posições de `fsp1` que ocorram em `fsp2`.

2.12 Sequências de variáveis

Se nós temos duas regras num sistema não ambíguo cujo lado esquerdo estão em posições não paralelas, nós sabemos da existência de uma variável que se interpõe da forma que será apresentada aqui.

Porém, mais do que isso, nós temos duas sequências de regras. Há a possibilidade de que o lado esquerdo de várias regras ocorram no lado esquerdo de outra. Assim, precisamos organizar as muitas variáveis, fruto das muitas sobreposições, em uma sequência. As quatro aplicações a seguir nos ajudarão nisso.

```
set_var(sigma,e,t): set[(V)] =
         {x:(V)| subterm(t,sigma(x)) AND Vars(lhs(e))(x)}

seq_var(E:Orthogonal,s,(p:positions?(s)),sigma,
      (e|member(e,E) & subtermOF(s,p)=ext(sigma)(lhs(e))),
      (fsp:SPP(s)|(FORALL(i:below[fsp`length]):p<=fsp`seq(i))),
      fse|fsp`length=fse`length & (FORALL (i:below[length(fse)]) :
      member(fse(i),E)), fss|fsp`length=fss`length &
      (FORALL (i:below[length(fss)]) :
      subtermOF(s,fsp`seq(i))=ext(fss`seq(i))(lhs(fse`seq(i))))):
      RECURSIVE finseq[(V)] =
       IF length(fsp)=0 THEN empty_seq
         ELSIF nonempty?(set_var(sigma,e,ext(fss`seq(0))(lhs(fse`seq(0)))))
          THEN add_first(choose(set_var(sigma,e,ext(fss`seq(0))(lhs(fse`seq(0))))),
                         seq_var(E,s,p,sigma,e,rest(fsp),rest(fse),rest(fss)))
          ELSE seq_var(E,s,p,sigma,e,rest(fsp),rest(fse),rest(fss))
      ENDIF
      MEASURE(length(fsp))
```

O conjunto `set_var(sigma,e,t)` contém as variáveis tais que `t` seja um subtermo da instanciação dessas variáveis pela substituição `sigma` e tais que ocorram dentro do lado esquerdo da regra `e`. Resumindo, caso o termo `t` seja subtermo de `lhs(e)`, queremos selecionar as variáveis cuja instanciação por uma substituição predefinida esteja numa posição intermediária entre os dois termos. No nosso caso, como as regras trabalhadas são lineares à esquerda, se o conjunto formado for não vazio, então será unitário.

O conceito anterior é importante para trabalharmos a definição de `seq_var`, que é muito mais complexa e as condições sobre os argumentos estão mais amarradas. Primeiramente, consideremos cada entrada dessa função: um conjunto de regras ortogonais `E`; um termo `s`; uma posição `p` de `s`; uma substituição `sigma`; uma regra `e`∈`E` tal que o subtermo de `s` em `p` é a instaciação do lado esquerdo de `e` por `sigma`; uma sequência `fsp` de posições paralelas de `s`, tais que as posições têm `p` como prefixo

(estão por baixo de `p`); uma sequência `fse` de regras em `E` com o mesmo comprimento de `fsp`; uma sequência `fss` de substituições com o mesmo comprimento de `fsp` tal que, para cada índice da sequência, o subtermo de `s` na posição em `fsp` é uma instanciação do lado esquerdo da regra em `fse` pela substituição em `fss`. A ideia desse construtor é a de que, para cada subtermo formado por `fss` e `fse` nas posições em `fsp`, toma-se uma variável tirada do conjunto `set_var` com argumentos `sigma`, `e` e cada subtermo. Sabemos que esses conjuntos são não vazios, pois todas as regras tratadas pertencem ao sistema não ambíguo `E`. Por outro lado, a sequência `seq_var` é unicamente determinada pois cada um desses conjuntos é unitário, uma vez que a regra `e` é linear à esquerda.

```
seq_var_par(E:Orthogonal,s,(p:positions?(s)),sigma,
       (e|member(e,E) & subtermOF(s,p)=ext(sigma)(lhs(e))),
       (fsp:SPP(s)|FORALL(i:below[fsp‘length]):p<=fsp‘seq(i)),
       fse|fsp‘length=fse‘length & (FORALL (i:below[length(fse)]) :
       member(fse(i),E)), fss|fsp‘length=fss‘length &
       (FORALL (i:below[length(fss)]) :
       subtermOF(s,fsp‘seq(i))=ext(fss‘seq(i))(lhs(fse‘seq(i))))):
      RECURSIVE finseq[(V)] =
          IF length(fsp)=0 THEN empty_seq
             ELSIF (FORALL (i: below[length(fsp)]):
                   i/=0 & seq_var(E,s,p,sigma,e,fsp,fse,fss)‘seq(0)/=
                                  seq_var(E,s,p,sigma,e,fsp,fse,fss)‘seq(i))
               THEN add_first(seq_var(E,s,p,sigma,e,fsp,fse,fss)‘seq(0),
                    seq_var_par(E,s,p,sigma,e,rest(fsp),rest(fse),rest(fss)))
               ELSE seq_var_par(E,s,p,sigma,e,rest(fsp),rest(fse),rest(fss))
             ENDIF
       MEASURE(length(fsp))
```

`seq_var_par` utiliza os mesmos argumentos que `seq_var` e terá como resultado uma sequência com as mesmas variáveis que resultam da outra, diferindo pelo fato de eliminar repetições de variáveis.

```
SEQ_VAR_multi(E:Orthogonal,s,(fsp1:SPP(s)),fss1|fss1‘length=fsp1‘length,
       (fse1|FORALL(i:below[fse1‘length]):member(fse1‘seq(i),E)
        & subtermOF(s,fsp1‘seq(i))=ext(fss1‘seq(i))(lhs(fse1‘seq(i)))),
       (fsp2:SPP(s)),fse2|fsp2‘length=fse2‘length &
       (FORALL (i:below[length(fse2)]) :
       member(fse2(i),E)), fss2|fsp2‘length=fss2‘length &
       (FORALL (i:below[length(fss2)]) :
       subtermOF(s,fsp2‘seq(i))=ext(fss2‘seq(i))(lhs(fse2‘seq(i))))):
      RECURSIVE finseq[finseq[(V)]] =
        IF length(fsp1)=0 THEN empty_seq
```

```
        ELSE add_first(seq_var_par(E,s,fsp1‘seq(0),fss1‘seq(0),fse1‘seq(0),
                       sub_pos(fsp2,fsp1‘seq(0)),
                       choose_seq(sub_pos(fsp2,fsp1‘seq(0)),fsp2,fse2),
                       choose_seq(sub_pos(fsp2,fsp1‘seq(0)),fsp2,fss2)),
                  SEQ_VAR_multi(E,s,rest(fsp1),rest(fss1),rest(fse1),
                                fsp2,fse2,fss2))
      ENDIF
    MEASURE(length(fsp1))
```

O algoritmo `SEQ_VAR_multi` cria uma sequência de sequências de variáveis. Toma-se o primeiro argumento das sequências `fsp1` de posições, `fss1` de substituições e `fse1` de regras e aplica-se a `seq_var_par` (`fsp1'seq(0)` como a posição, `fse1'seq(0)` como a regra e `fss1'seq(0)` como a substituição) de forma que se cumpram as condições exigidas anteriormente. Para que a condição de subtermo seja cumprida também, como sequência de posições a aplicar-se a `seq_var_par`, tomamos `sub_pos(fsp2,fsp1‘seq(0))` e, para as sequências de regras e substituições, utilizamos `choose_seq` para ajustar `fse2` e `fss2` a `sub_pos`.

2.13 Tratamento de sequências de posições

Quando lidamos com duas reduções em paralelo sobre o mesmo termo, formando uma divergência, algumas posições de uma das sequências de posições podem estar por baixo de posições da outra sequência. Como em cada posição, nós temos um termo que faz *matching* com o lado esquerdo de uma regra dentro de um sistema ortogonal, mesmo que instanciado por uma substituição, pela seção anterior sabemos que existem variáveis que se interpõem se estiverem em posições não paralelas. No entanto, após a reescrita, as variáveis podem assumir diferentes posições no lado direito da regra que a contém. Assim, os subtermos das variáveis também se deslocarão.

Os próximos construtores tratam as posições dessas sequências desmembrando as posições de variáveis e concatenando com as novas no lado direito das regras. Vejamos como isso acontece passo a passo.

```
complement_pos_set(p1,p): set[position] =
        IF p1 <= p & p1 /= p
          THEN {p2 | p=p1 o p2}
          ELSE emptyset
        ENDIF

complement_pos(p,(fsp:PP)):
      RECURSIVE finseq[position] =
          IF length(fsp)=0 THEN empty_seq
```

```
          ELSIF nonempty?(complement_pos_set(p,fsp`seq(0)))
             THEN add_first(choose(complement_pos_set(p,fsp`seq(0))),
                            complement_pos(p,rest(fsp)))
             ELSE complement_pos(p,rest(fsp))
          ENDIF
       MEASURE(length(fsp))
```

Temos aqui o conjunto `complement_pos_set(p1,p)`, que seleciona o complemento de `p1` até `p`, caso `p1` seja prefixo de `p` e as duas posições não coincidam. Obviamente, se estas duas condições são atendidas, o conjunto será unitário; caso contrário, será vazio. Já a sequência `complement_pos(p,fsp)` é formada pelo complemento das posições de `fsp` em relação a `p`, caso eles existam. Assim, `complement_pos` é o nosso instrumento para cortar posições.

```
compo_pos(p,(fsp:PP)): RECURSIVE finseq[position] =
          IF length(fsp)=0 THEN empty_seq
             ELSE add_first(p o fsp`seq(0),compo_pos(p,rest(fsp)))
          ENDIF
       MEASURE(length(fsp))

compo_pos_multi(fsp1,fsp2:PP): RECURSIVE finseq[position] =
          IF length(fsp1)=0 THEN empty_seq
             ELSE compo_pos(fsp1`seq(0),fsp2) o
                            compo_pos_multi(rest(fsp1),fsp2)
          ENDIF
       MEASURE(length(fsp1))
```

`compo_pos` fará o contrário, concatenando a posição `p` a cada uma em `fsp`. Esse procedimento é repetidamente realizado em `compo_pos_multi`, que concatena cada posição de `fsp1` a todas as posições de `fsp2`.

```
compo_pos_var(s,t,fsv,p,(fsp:PP)): RECURSIVE finseq[position] =
          IF length(fsv)=0 THEN empty_seq
             ELSIF nonempty?(Pos_var(s,fsv`seq(0)))
                  THEN compo_pos(p,compo_pos_multi(set2seq(Pos_var(t,fsv`seq(0))),
                           complement_pos(choose(Pos_var(s,fsv`seq(0))),
                                          complement_pos(p,fsp)))) o
                    compo_pos_var(s,t,rest(fsv),p,fsp)
                  ELSE compo_pos_var(s,t,rest(fsv),p,fsp)
          ENDIF
       MEASURE(length(fsv))
```

```
Compo_pos_var(fst1:finseq[term],(fst2:finseq[term]|fst1‘length=fst2‘length),
               FSV:finseq[finseq[(V)]]|fst1‘length=FSV‘length,
               fsp1:PP|fsp1‘length=fst1‘length,
              (fsp2:PP)):
       RECURSIVE finseq[position] =
          IF length(fsp1)=0 THEN empty_seq
             ELSE add_first(compo_pos_var(fst1‘seq(0),fst2‘seq(0),FSV‘seq(0),
                                          fsp1‘seq(0),fsp2),
                            Compo_pos_var(rest(fst1),rest(fst2),rest(FSV),
                                          rest(fsp1),rest(fsp2)))
          ENDIF
       MEASURE(length(fsp1))
```

Nesse momento, os dois tipos de operações são realizadas: o corte e a composição de posições. Para cada variável na sequência `fsv`, caso esta ocorra no termo `s`, a função `compo_pos_var` toma o complemento de uma posição da variável em `s` até o complemento de `p` a uma sequência de posições paralelas `fsp`. Teremos em mãos o complemento das posições de `fsp` dentro da variável. Depois disso, concatenaremos essas posições curtas às posições da variável no outro termo `t` e, por fim, compomos com a posição `p` novamente.

```
compo_rr_var(s,t,fsv,p,(fsp:PP),fse|length(fsp)=length(fse)):
  RECURSIVE finseq[rewrite_rule] =
  IF length(fsv)=0 THEN empty_seq
    ELSIF nonempty?(Pos_var(s,fsv‘seq(0)))
         THEN power(choose_seq(sub_pos(fsp,
                                       compo_pos(p,set2seq(Pos_var(s,fsv‘seq(0))))‘seq(0)),
                               fsp,fse),
                    card(Pos_var(t,fsv‘seq(0))))
              o compo_rr_var(s,t,rest(fsv),p,fsp,fse)
         ELSE compo_rr_var(s,t,rest(fsv),p,fsp,fse)
  ENDIF
 MEASURE(length(fsv))

compo_Sub_var(s,t,fsv,p,(fsp:PP),fss|length(fsp)=length(fss)):
  RECURSIVE finseq[Sub] =
  IF length(fsv)=0 THEN empty_seq
    ELSIF nonempty?(Pos_var(s,fsv‘seq(0)))
         THEN power(choose_seq(sub_pos(fsp,
                                       compo_pos(p,set2seq(Pos_var(s,fsv‘seq(0))))‘seq(0)),
                               fsp,fss),
                    card(Pos_var(t,fsv‘seq(0))))
              o compo_Sub_var(s,t,rest(fsv),p,fsp,fss)
```

```
        ELSE compo_Sub_var(s,t,rest(fsv),p,fsp,fss)
  ENDIF
 MEASURE(length(fsv))
```

Quando nós tomamos sequências de posições formadas por construtores como `Pos_Over` ou `sub_pos`, que apenas selecionam posições específicas dentro das originais, é fácil ajustar as sequências de substituições e regras que as acompanham através de `choose_seq`. Porém, isso não vale para `compo_pos_var`, pois as posições na sequência resultante são diferentes do que tínhamos nas originais. Assim, criamos as aplicações `compo_rr_var` e `compo_sub_var` que fazem o ajuste necessário para este tipo específico de sequência de posições, fazendo com que as sequências de regras e substituições criadas pelos respectivos construtores acompanhem os índices da sequência de posições criada por `compo_pos_var`.

Nós temos os mesmos argumentos de `compo_pos_var` acrescidos de uma sequência `fse` de regras de reescrita para o caso de `compo_rr_var` e `fss` de substituições para o caso de `compo_Sub_var`, em ambos os casos com o mesmo comprimento da sequência de posições `fsp`.

Nós sabemos que `complement_pos(p,fsp)` e `sub_pos(fsp,p)` selecionam as mesmas posições, com a diferença que o primeiro as quebra e o segundo não. Isso fará toda a diferença, pois usando `sub_pos`, poderemos nos utilizar de `choose_seq` pelos motivos supra mencionados.

Para cada variável em `fsv`, tomamos as subposições de `fsp` em relação à uma posição da variável em `s`. Daí, ajustamos as regras/substituições que acompanham essas subposições através de `choose_seq`. Depois, nós replicamos essa sequência de regras/substituições pelo número de ocorrência da variável no termo `t` usando a aplicação `power`.

```
Compo_rr_var(fst1,(fst2|fst1`length=fst2`length),
               FSV|fst1`length=FSV`length,fsp1:PP|fsp1`length=fst1`length,
              (fsp2:PP),fse|length(fsp2)=length(fse)):
       RECURSIVE finseq[rewrite_rule] =
          IF length(fsp1)=0 THEN empty_seq
             ELSE compo_rr_var(fst1`seq(0),fst2`seq(0),
                               FSV`seq(0),fsp1`seq(0),fsp2,fse)
                  o Compo_rr_var(rest(fst1),rest(fst2),
                                 rest(FSV),rest(fsp1),fsp2,fse)
          ENDIF
       MEASURE(length(fsp1))

Compo_Sub_var(fst1,(fst2|fst1`length=fst2`length),
```

```
            FSV|fst1‘length=FSV‘length,fsp1:PP|fsp1‘length=fst1‘length,
           (fsp2:PP),fss|length(fsp2)=length(fss)):
      RECURSIVE finseq[Sub] =
         IF length(fsp1)=0 THEN empty_seq
            ELSE compo_Sub_var(fst1‘seq(0),fst2‘seq(0),
                               FSV‘seq(0),fsp1‘seq(0),fsp2,fss)
                 o Compo_Sub_var(rest(fst1),rest(fst2),
                                 rest(FSV),rest(fsp1),fsp2,fss)
         ENDIF
      MEASURE(length(fsp1))
```

As funções `Compo_rr_var` e `Compo_Sub_var` fazem o mesmo que `Compo_pos_var`, aplicando respectivamente `compo_rr_var` e `compo_Sub_var`, aos termos da sequência `fst1` como primeiro termo, os da sequência `fst2` como segundo termo e a cada sequência de variáveis dentro de `FSV`.

```
Pos_seq_var(s,fsv): RECURSIVE finseq[position]=
          IF length(fsv)=0 THEN empty_seq
             ELSE set2seq(Pos_var(s,fsv‘seq(0))) o
                  Pos_seq_var(s,rest(fsv))
          ENDIF
       MEASURE(length(fsv))
```

Durante a prova será importante considerar as posições onde variáveis ocorrem, não uma a uma (pois já temos o recurso `Pos_var`), mas de todas ao mesmo tempo. Para tanto, temos `Pos_seq_var(s, fsv)`, que nos dá a sequência formada pelas posições onde cada variável da sequência `fsv` ocorre no termo `s`.

2.14 Sequências de termos

Ao longo da especificação nós abordamos e ainda abordaremos sequências de termos como argumento nas nossas funções. No entanto, quando tratamos de redução em paralelo, as únicas sequências que surgem são as de posições, substituições e regras de reescrita. Torna-se fundamental, então, construir tais sequências de termos para que as nossas ferramentas criadas até agora se tornem úteis.

```
left_rewrite_rules(fse): RECURSIVE finseq[term]=
          IF length(fse)=0 THEN empty_seq
             ELSE add_first(lhs(fse‘seq(0)),left_rewrite_rules(rest(fse)))
          ENDIF
       MEASURE(length(fse))
```

```
right_rewrite_rules(fse): RECURSIVE finseq[term]=
          IF length(fse)=0 THEN empty_seq
             ELSE add_first(rhs(fse`seq(0)),right_rewrite_rules(rest(fse)))
          ENDIF
       MEASURE(length(fse))

Subs_right_rr(fse,(fss|fse`length=fss`length)): RECURSIVE finseq[term]=
          IF length(fse)=0 THEN empty_seq
             ELSE add_first(ext(fss`seq(0))(rhs(fse`seq(0))),
                            Subs_right_rr(rest(fse),rest(fss)))
          ENDIF
       MEASURE(length(fse))
```

Os nossos construtores aqui são extremamente simples. O primeiro é `left_rewrite_rules`, que é aplicado a uma sequência de regras de reescrita e retorna uma sequência com o termo do lado esquerdo de cada uma. Do mesmo modo, `right_rewrite_rules` retorna o lado direito das regras. Já a aplicação `Subs_right_rr` é aplicado a duas sequências: uma de regras e outra de substituições, as duas com o mesmo comprimento. Toma-se o termo do lado direito de cada regra e instancia-se com a substituição correspondente.

2.15 Sequências de substituições

Quando nós realizamos uma redução em paralelo, existem sequências de posições, regras e substituições associadas entre si. Mas, quando nós consideramos duas reduções em paralelo sobre o mesmo termo, pode existir a sobreposição das posições trabalhadas, sendo que existem regras instanciadas cujo lado esquerdo está dentro do lado esquerdo de uma outra regra instanciada. Depois que a reescrita é feita nas subposições, é necessário realizar a reescrita sobre a regra que as contém (o termo maior). Todavia, a antiga substituição que a instanciava antes não serve mais, pois alguns subtermos foram modificados. Assim, precisamos criar uma nova substituição que dê conta de criar o *matching* do termo maior com o lado esquerdo da regra instanciada com essa substituição atualizada.

Vamos agora considerar como viabilizaremos isso.

```
replace_terms(s,fst,(fsp:SPP(s)|fsp`length=fst`length)): RECURSIVE term =
        IF length(fst)=0 THEN s
           ELSE replace_terms(replaceTerm(s,fst`seq(0),fsp`seq(0)),rest(fst),rest(fsp))
        ENDIF
     MEASURE(fst`length)
```

```
SIGMA(sigma, x, fst, (fsp:SPP(sigma(x))|length(fsp)=length(fst)))(y:(V)):
    term = IF length(fst)=0 OR y/=x
           THEN sigma(y)
           ELSE replace_terms(sigma(x),fst,fsp)
        ENDIF
```

O recurso `replace_terms` nos permite trocar todos os subtermos de `s` nas posições paralelas em `fsp` pelos termos da sequência `fst`. A aplicação `SIGMA` utilizará o último meio de forma que, quando aplicamos `SIGMA(sigma,x,fst,fsp)` a uma variável `y`, teremos a mesma ação de `sigma` a menos que `y` seja igual a `x`. Nesse caso, trocamos todos os subtermos de `sigma(x)` nas posições em `fsp` pelos termos em `fst`.

```
SIGMAP(sigma,fsv,(fsp1:PP|fsp1`length=fsv`length),
       fst,(fsp2:PP|fsp2`length=fst`length))(y:(V)):
     RECURSIVE term=
        IF length(fsv)=0 THEN sigma(y)
           ELSIF y=fsv`seq(0) &
                 SP?(sigma(fsv`seq(0)))(complement_pos(fsp1`seq(0),fsp2))
              THEN SIGMA(sigma,fsv`seq(0),
                         choose_seq(sub_pos(fsp2,fsp1`seq(0)),fsp2,fst),
                         complement_pos(fsp1`seq(0),fsp2))(y)
              ELSE SIGMAP(sigma,rest(fsv),rest(fsp1),fst,fsp2)(y)
        ENDIF
     MEASURE(length(fsv))
```

`SIGMAP` faz o trabalho de `SIGMA` em cada variável da sequência `fsv`. Para ajustar as condições sobre a sequência de posições para que estejam dentro da `sigma`-instanciação de cada variável, tomamos as posições de `fsp2` que tenham complemento em relação às posições em `fsp1`. A ideia é que cada entrada em `fsp1` seja posição da variável em um termo maior. Por enquanto, essa situação não está explícita, mas quando estivermos formalizando as provas, será necessário ter as condições requeridas nos argumentos de `SIGMA`.

```
SIGMA_multi(fsp1:PP,(fss1|fsp1`length=fss1`length),
            (FSV:finseq[finseq[(V)]]|fsp1`length=FSV`length &
             FORALL(i:below[FSV`length]):
             FORALL(m,n:below[(FSV`seq(i))`length]):m/=n
               IMPLIES (FSV`seq(i))`seq(m)/=(FSV`seq(i))`seq(n)),
            (fse1|fsp1`length=fse1`length & (FORALL(i:below[fse1`length]):
             Pos_seq_var(lhs(fse1`seq(i)),FSV`seq(i))`length=(FSV`seq(i))`length)),
```

```
       fst,(fsp2:PP|fsp2`length=fst`length)):
RECURSIVE finseq[Sub]=
   IF length(fsp1)=0 THEN empty_seq
      ELSE add_first(SIGMAP(fss1`seq(0),FSV`seq(0),
                     Pos_seq_var(lhs(fse1`seq(0)),FSV`seq(0)),
                     choose_seq(sub_pos(fsp2,fsp1`seq(0)),fsp2,fst),
                     complement_pos(fsp1`seq(0),fsp2)),
                     SIGMA_multi(rest(fsp1),rest(fss1),
                                 rest(FSV),rest(fse1),fst,fsp2))
   ENDIF
MEASURE(length(fsp1))
```

Agora temos uma sequência de substituições criada por `SIGMA_multi`. Esse construtor ordenará a aplicação de `SIGMAP` sobre as substituições em `fss1`, as sequências de variáveis em `FSV` e as posições dessas sequências de variáveis dentro do lado esquerdo de cada regra em `fse1`.

Isso completa a nossa especificação das definições que serão nossas ferramentas na formalização que desenvolveremos no próximo capítulo.

Capítulo

3

Formalização

O assistente de provas PVS pode ser utilizado através dos comandos de prova apresentados na seção 1.1. Abre-se cada teorema num ambiente de demonstração que organiza as proposições como negativas quando as tomamos como premissas e positivas quando são teses, da formada apresentada a seguir:

```
[-1]  x=9
[-2]  f(x,a) <= 0
[-3]  ...
      ...
  |-------
[1]   4 > 9
[2]   f(9,a) = 0
[3]   ...
      ...
```

Como vimos anteriormente, assume-se a conjunção dos itens negativos e busca-se provar a disjunção da parte positiva, ou seja, assumindo todas as premissas, precisa-se provar alguma tese. Isso é obtido quando podemos aplicar uma regra axiomática, isto é, quando ocorre um dos cinco casos:

1. uma premissa coincide com uma tese.
   Ex: `[-2] x + 3 = 9`

   ```
      |---

   [1]   x = 6
   ```

2. uma premissa é obviamente falsa.
   Ex: `[-1] 0 > 3`

3. as premissas se contradizem
   Ex: `[-1] a <= 8`

```
[-2] a = b

[-3] b > 12
```

4. uma tese é obviamente verdadeira
   Ex: `[3] 12 <= 12`

5. a união de duas ou mais teses englobam todas as possibilidades para um dado objeto
   Ex: `[1] n <= 1`

   `[2] 2 <= n`

Isso nos deixa prontos para falar a respeito da formalização do teorema que mostra que TRS's ortogonais são confluentes. Aqui seguiremos a estratégia apresentada na seção 1.2.3. Como vimos, a ideia principal é provar a propriedade do diamante para a redução em paralelo (`parallel_reduction?(E)`) para um sistema ortogonal. Como a confluência é inferida da propriedade diamante, provando a equivalência entre o fecho reflexivo transitivo da redução em paralelo e da redução usual, temos a prova que procurávamos.

3.1 RTC(reduction?(E)) = RTC(parallel_reduction?(E))

Para demonstrar a enquivalência entre as relações `RTC(reduction?(E))` e `RTC(parallel_reduction?(E)`, vamos demonstrar primeiramente as inclusões de `reduction?(E)` em `parallel_reduction?(E)` e desta em `RTC(reduction?(E))`. Isso é expressado pelo lema abaixo, que diz que, para quaisquer dois termos `t1` e `t2` valem as inclusões mencionadas.

```
parallel_reduction: LEMMA
 (reduction?(E)(t1, t2) => parallel_reduction?(E)(t1, t2))
  & (parallel_reduction?(E)(t1, t2) => RTC(reduction?(E))(t1, t2))
```

A prova quebra-se em duas partes através do comando `split` para que se prove cada uma das inclusões. A primeira é mais tranquila de ser provada, como veremos. De fato, nós temos:

```
parallel_reduction.1 :

{-1}  reduction?(E)(t1, t2)
  |-------
{1}   parallel_reduction?(E)(t1, t2)
```

Expandindo-se a definição de `reduction?(E)` com o comando `expand` nós inferimos a existência de uma posição, uma regra e uma substituição com as quais se realiza a redução de `t1` para `t2`. Por outro lado, ao expandirmos a definição de `parallel_reduction?(E)`, precisamos fornecer uma sequência de posições, uma de regras e uma de substituições que realizam a redução dos mesmos termos. Então, nós temos:

```
parallel_reduction.1 :

{-1}  EXISTS ((e: rewrite_rule[variable, symbol, arity] | member(e, E)),
              sigma: Sub[variable, symbol, arity],
              (p: positions?[variable, symbol, arity](t1))):
        subtermOF(t1, p) = ext(sigma)(lhs(e)) &
         t2 = replaceTerm(t1, ext(sigma)(rhs(e)), p)
  |-------
{1}   EXISTS (fsp: SPP(t1),
              fse: finseq[rewrite_rule]
                  | FORALL (i: below[fse`length]): member(fse`seq(i), E),
              fss):
        fsp`length = fse`length AND
         fsp`length = fss`length AND
          (FORALL (i: below[fsp`length]):
             subtermOF(t1,
                       finseq_appl[position[variable, symbol, arity]]
                           (fsp)(i))
              =
              ext(finseq_appl[Sub[variable, symbol, arity]](fss)(i))
                 (lhs(finseq_appl[rewrite_rule[variable, symbol, arity]]
                           (fse)(i))))
           AND t2 = replace_par_pos(t1, fsp, fse, fss)
```

Depois aplicamos o comando `skosimp` que substitui as variáveis quantificadas existencialmente por constantes em {-1} com a regra `e!1`, a substituição `sigma!1` e a posição `p!1`. Daí podemos instanciar {1} com a sequência unitária de posição `(# length := 1, seq := (LAMBDA (n: below[1]): p!1) #)` (de comprimento 1 e termos iguais a `p!1`), de regra `(# length := 1, seq := (LAMBDA (n: below[1]): e!1) #)` e de substituição `(# length := 1, seq := (LAMBDA (n: below[1]): sigma!1) #)`.

No entanto, ao instanciarmos com essas sequências, devemos estar certos de que elas cumprem as condições para a redução em paralelo. Daí a prova se divide em dois casos o primeiro onde assumimos que a sequência unitária de posição é de posições paralelas do termo `t1` e o segundo onde provamos essa tal propriedade. No primeiro caso, temos a seguinte situação:

```
parallel_reduction.1.1 :

[-1]  subtermOF(t1, p!1) = ext(sigma!1)(lhs(e!1))
[-2]  t2 = replaceTerm(t1, ext(sigma!1)(rhs(e!1)), p!1)
  |-------
{1}   (FORALL (i: below[1]):
         subtermOF(t1,
                   finseq_appl[position[variable, symbol, arity]]
                       ((# length := 1,
                           seq := (LAMBDA (n: below[1]): p!1) #))
                       (i))
          =
          ext(finseq_appl[Sub[variable, symbol, arity]]
                  ((# length := 1,
                      seq := (LAMBDA (n: below[1]): sigma!1) #))
                  (i))
             (lhs(finseq_appl[rewrite_rule[variable, symbol, arity]]
                       ((# length := 1,
                           seq := (LAMBDA (n: below[1]): e!1) #))
                       (i))))
       AND
       t2 =
        replace_par_pos(t1,
                        (# length := 1,
                           seq := (LAMBDA (n: below[1]): p!1) #),
                        (# length := 1,
                           seq := (LAMBDA (n: below[1]): e!1) #),
                        (# length := 1,
                           seq := (LAMBDA (n: below[1]): sigma!1) #))
```

Este caso se resolve trivialmente usando o comando `grind`, que substitui as variáveis quantificadas existencialmente por constantes e expande as definições necessárias. Daí, ele se dá conta que a única posição a se considerar é `p!1`, a única regra é `e!1` e a única substituição é `sigma!1`, além do fato que `replace_par_pos` equivale a `replaceTerm` quando as sequências são unitárias. Vamos ao segundo caso.

```
parallel_reduction.2 :

{-1}  parallel_reduction?(E)(t1, t2)
  |-------
{1}   RTC(reduction?(E))(t1, t2)
```

Como `parallel_reduction?` utiliza a aplicação `replace_par_pos`, que é construída recursivamente sobre o comprimento de uma sequência `fsp` de posições, desejamos fazer uma prova por indução sobre essa medida. No entanto, quando expandimos a definição de `parallel_reduction?` apresentada na Seção 2.10, `fsp` aparece quantificada existencialmente. Isso nos impede de aplicarmos o método de indução diretamente. Assim, supomos uma proposição onde `fsp` aparece quantificada universalmente, a partir da qual seja trivial obtermos a nossa prova. Assim, utilizamos o comando

```
(case "FORALL (t: term, fsp: SPP(t1), fse: finseq[rewrite_rule] |
               (FORALL (i: below[fse`length]): member(fse`seq(i), E)),
               fss: finseq[Sub]):
                     (fsp`length = fse`length & fsp`length = fss`length &
                     (FORALL (i: below[fsp`length]): positionsOF(t)(fsp`seq(i))
                             & subtermOF(t1, fsp`seq(i))
                                       = ext(fss`seq(i))(lhs(fse`seq(i)))
                             & subtermOF(t, fsp`seq(i))
                                       = ext(fss`seq(i))(lhs(fse`seq(i)))) &
                      t2 = replace_par_pos(t, fsp, fse, fss))
                           => RTC(reduction?(E))(t, t2)")
```

De fato, quando expandimos a definição de `parallel_reduction?`, supõe-se a existência de sequências de posições, regras e substituições que cumprem as hipóteses do caso acima. Então, skolemizamos essa definição com as sequências `fsp!1`, `fse!1` e `fss!1` e podemos instanciar o caso quantificado universalmente com o termo `t1` e as sequências obtidas. Daí, obtemos o resultado trivialmente, pois a tese do caso é igual à tese do sequente (`RTC(reduction?(E))(t, t2)`).

Por outro lado, temos o caso quantificado universalmente como tese. Assim, podemos aplicar indução sobre o comprimento da sequência `fsp` de posições através do comando `(measure-induct+ "fsp`length" "fsp")`, que skolemiza a sequência com o nome `x!1`. Primeiramente, supomos o caso em que o comprimento de `x!1` é igual a zero e escondemos as proposições desnecessárias aqui.

```
parallel_reduction.2.2.1.1 :

[-1]  length(x!1) = 0
[-2]  x!1`length = fse!1`length
[-3]  x!1`length = fss!1`length
[-4]  t2 = replace_par_pos(t!1, x!1, fse!1, fss!1)
  |-------
[1]   RTC(reduction?(E))(t!1, t2)
```

Expandindo a definição de `replace_par_pos`, temos que `t2 = t!1`. Por outro lado, expandindo `RTC`, precisamos mostrar que existe um número de iterações de `reduction?(E)` em que `t!1` atinge `t2`. Isso é trivial, pois `t!1` atinge `t2` em 0 passos.

Agora, temos o oposto: a sequência `x!1` de posições não tem comprimento zero. O objetivo é provar que a propriedade vale para `x!1` supondo que a propriedade vale para uma sequência com comprimento menor que o de `x!1`.

```
parallel_reduction.2.2.1.2 :

[-1]  FORALL (y: SPP[variable, symbol, arity](t1)):
        FORALL (t: term,
                fse: finseq[rewrite_rule]
                    | (FORALL (i: below[fse`length]):
                         member(fse`seq(i), E)),
                fss: finseq[Sub]):
          y`length < x!1`length IMPLIES
           (y`length = fse`length &
             y`length = fss`length &
              (FORALL (i: below[y`length]):
                 positionsOF(t)(y`seq(i)) &
                  subtermOF(t1, y`seq(i)) =
                   ext(fss`seq(i))(lhs(fse`seq(i)))
                   &
                   subtermOF(t, y`seq(i)) =
                    ext(fss`seq(i))(lhs(fse`seq(i))))
             & t2 = replace_par_pos(t, y, fse, fss))
            => RTC(reduction?(E))(t, t2)
[-2]  x!1`length = fse!1`length
[-3]  x!1`length = fss!1`length
[-4]  FORALL (i: below[x!1`length]):
        positionsOF(t!1)(x!1`seq(i)) &
         subtermOF(t1, x!1`seq(i)) = ext(fss!1`seq(i))(lhs(fse!1`seq(i))) &
          subtermOF(t!1, x!1`seq(i)) = ext(fss!1`seq(i))(lhs(fse!1`seq(i)))
[-5]  t2 = replace_par_pos(t!1, x!1, fse!1, fss!1)
  |-------
[1]   length(x!1) = 0
[2]   RTC(reduction?(E))(t!1, t2)
```

Então, instanciamos `[-1]` com `rest(x!1)`, `replaceTerm(t!1, ext(fss!1`seq(0))(rhs(fse!1`seq(0))), x!1`seq(0))`, `rest(fse!1)` e `rest(fss!1)` e aplicamos o comando `(prop)` que quebrará a prova em vários casos, sendo o primeiro supondo a tese

---

`RTC(reduction?(E))(replaceTerm(t!1, ext(fss!1‘seq(0)) (rhs(fse!1‘seq(0))), x!1‘seq(0)), t2)` e os outros são as premissas da proposição `[-1]`. As propriedades principais que precisamos provar é que a composição `reduction?(E) o RTC(reduction?(E))` é igual a `RTC(reduction?(E))` e que `replaceTerm` comuta com `replace_par_pos` desde que as posições de trocas sejam paralelas.

Então, nós concluimos a prova anterior e partimos para a próxima, que é provar a equivalência propriamente dita entre os fechos reflexivo transitivos de `reduction?(E!1)` e de `parallel_reduction?(E!1)`.

```
parallel_reduction_RTC :

  |-------
{1}   RTC(reduction?(E!1)) = RTC(parallel_reduction?(E!1))
```

Para provarmos essa equivalência, na verdade precisamos mostrar que ela vale para qualquer par de termos. Assim, nós podemos introduzir um par de termos arbitrários para que se possa provar a igualdade através do comando `(decompose-equality)`. Daí, teríamos a igualdade de proposições `RTC(reduction?(E!1))(x!1, x!2) = RTC(parallel_reduction?(E!1))(x!1, x!2)`, que quer dizer que uma só acontece se, e somente se, a outra acontece. Então, podemos transformar a igualdade no conectivo lógico `IFF` através do comando `(iff)`. Depois disso, podemos quebrar a prova em dois casos principais com `(split)`, demonstrando as duas implicações.

```
parallel_reduction_RTC.1 :

{-1}  RTC(reduction?(E!1))(x!1, x!2)
  |-------
{1}   RTC(parallel_reduction?(E!1))(x!1, x!2)
```

Sabemos que `RTC` supõe a existência de um número de iterações da relação correspondente em que o primeiro termo atinge o segundo. O nosso objetivo é fazer uma prova indutiva, mas como no outro lema, a quantificação existencial nos impede de assim o fazer. Logo, assumimos um caso onde o número de passos esteja quantificado universalmente e pelo qual seja óbvio atingir a nossa prova.

Nesse primeiro ramo, o caso assumido é `(case "FORALL (t1, t2: term, i: nat): iterate(reduction?(E!1), i)(t1, t2) => RTC(parallel_reduction?(E!1))(t1, t2)")`. Aplicamos, então, indução sobre `i`.

```
[-1]  iterate(reduction?(E!1), 0)(t1, t2)
  |-------
[1]   RTC(parallel_reduction?(E!1))(t1, t2)
```

Na base de indução, nós temos que provar que um termo `t1` que atinge `t2` em zero passos de `reduction?(E!1)` deve atingir `t2` também em algum número de passos de `parallel_reduction?(E!1)`. Mas isso é fácil de provar pois `[-1]` mostra que `t1 = t2` e, assim, `t1` atinge `t2` em zero passos de `parallel_reduction?(E!1)`.

```
parallel_reduction_RTC.1.2.2 :

[-1]  FORALL (t1, t2: term):
        iterate(reduction?(E!1), j)(t1, t2) =>
         RTC(parallel_reduction?(E!1))(t1, t2)
{-2}  iterate(reduction?(E!1), j + 1)(t1, t2)
  |-------
{1}   RTC(parallel_reduction?(E!1))(t1, t2)
```

Assumimos que podemos atingir `RTC(parallel_reduction?(E!1))` com `j` iterações de `reduction?(E!1)` e precisamos provar que o mesmo é conseguido com `j + 1` passos. Primeiramente, utilizamos o lema `iterate_add_one` que quebra a iteração de {-2} no primeiro passo de `reduction?(E!1)` concatenado com os outros `j` passos da relação. Isso significa que expusemos um termo `y!1` na cadeia entre `t1` e `t2`. Além disso, agora podemos utilizar a hipótese de indução e temos `RTC(parallel_reduction?(E!1))(y!1, t2)`.

```
parallel_reduction_RTC.1.2.2 :

{-1}  iterate(reduction?(E!1), j)(y!1, t2) =>
       RTC(parallel_reduction?(E!1))(y!1, t2)
[-2]  reduction?(E!1)(t1, y!1)
[-3]  iterate(reduction?(E!1), j)(y!1, t2)
  |-------
[1]   RTC(parallel_reduction?(E!1))(t1, t2)
```

No entanto, ainda não temos a propriedade para o par de termos `(t1, t2)`. O que nós podemos fazer é utilizar o lema `parallel_reduction`, que provamos anteriormente, e obter `parallel_reduction?(E!1)(t1, y!1))` de {-2}. Daí, utilizamos outro lema da biblioteca `ars` que mostra que um passo de uma relação composto com `RTC` da relação equivale ao `RTC` da relação.

Por outro lado, ao provar `RTC(parallel_reduction?(E!1))(x!1, x!2) IMPLIES RTC(reduction?(E!1))(x!1, x!2)`, nos utilizamos da mesma estratégia de supor um caso onde o número de iterações de `parallel_reduction?(E!1)` aparece instanciado universalmente. Usamos esse caso para fechar esse ramo da prova, mas temos de provar que ele é verdadeiro. Para tanto, aplicamos indução sobre esse número de iterações e obtemos o seguinte sequente:

```
parallel_reduction_RTC.2.2.2 :

[-1]  FORALL (t1, t2: term):
        iterate(parallel_reduction?(E!1), j)(t1, t2) =>
         RTC(reduction?(E!1))(t1, t2)
{-2}  iterate(parallel_reduction?(E!1), j + 1)(t1, t2)
  |-------
{1}   RTC(reduction?(E!1))(t1, t2)
```

Novamente, nós podemos explicitar o termo intermediário `y!1` que ocorre na cadeia de reduções entre os termos `t1` e `t2`. Assim, utilizando o lema `iterate_add_one` como no primeiro ramo da prova, nós obtemos `parallel_reduction?(E!1)(t1, y!1)` e `iterate(parallel_reduction?(E!1), j)(y!1, t2)`. Através do lema anterior, sabemos que `RTC(reduction?(E!1))(t1, y!1)`. Além disso, utilizando a hipótese de indução em `[-1]`, sabemos que `RTC(reduction?(E!1))(y!1, t2)`. Finalmente, pela transitividade de `RTC(reduction?(E!1))`, concluímos que `RTC(reduction?(E!1))(t1, t2)`.

Dessa forma, alcançamos o resultado que esperávamos de equivalência entre o fecho reflexivo transitivo das duas relações.

3.2 Orthogonal_implies_confluent

Na verdade, esse teorema é um corolário dos dois lemas anteriores e de um terceiro cuja prova será mostrada posteriormente (a redução em paralelo tem a propriedade do diamante).

A prova segue como no teorema 1.2.3.

```
Orthogonal_implies_confluent: LEMMA
 FORALL (E : Orthogonal) :
   LET RRE = reduction?(E) IN
   confluent?(RRE)
```

De fato, expandindo a definição de `confluent?`, nós temos que provar o seguinte sequente:

```
Orthogonal_implies_confluent :

[-1]  RTC(reduction?(E!1))(x, y)
[-2]  RTC(reduction?(E!1))(x, z)
  |-------
{1}   EXISTS (z_1: term[variable, symbol, arity]): RTC(reduction?(E!1))(y, z_1)
                                                  & RTC(reduction?(E!1))(z, z_1)
```

Nós usamos o lema `parallel_reduction_RTC` e obtemos que `RTC(reduction?(E!1)) = RTC(parallel_reduction?(E!1))`. Daí, provar a existência de tal termo `z_1` para a relação `reduction?(E!1)` equivale a prová-la para `parallel_reduction?(E!1)`.

Nós invocamos o lema `parallel_reduction_is_DP` em que a relação `parallel_reduction?(E!1)` tem a propriedade do diamante sempre que `E!1` for ortogonal.

No arcabouço encontrado em [1], já tínhamos os resultados `DP_implies_StC` e `Strong_Confl_implies_Confl`, ou seja, a propriedade do diamante implica em `parallel_reduction?(E!1)` fortemente confluente e isso implica em confluência, como queríamos.

```
Orthogonal_implies_confluent :

{-1}  RTC(parallel_reduction?(E!1))(y, z!1)
{-2}  RTC(parallel_reduction?(E!1))(z, z!1)
[-3]  strong_confluent?(parallel_reduction?(E!1))
[-4]  diamond_property?(parallel_reduction?(E!1))
[-5]  RTC(reduction?(E!1)) = RTC(parallel_reduction?(E!1))
[-6]  RTC(reduction?(E!1))(x, y)
[-7]  RTC(reduction?(E!1))(x, z)
  |-------
[1]   RTC(reduction?(E!1))(y, z!1) & RTC(reduction?(E!1))(z, z!1)
```

Expandindo a definição de confluência novamente, encontramos um termo `z!1` para o qual vale a propriedade e encerramos a prova com o comando `(assert)`.

3.3 parallel_reduction_is_DP

Se nós desejamos provar que a redução em paralelo tem a propriedade diamante, como vimos na seção 1.2.1, precisamos mostrar que, para uma dada divergência

`parallel_reduction?(E)(s,t1) & parallel_reduction?(E)(s,t2)`, onde `s`, `t1` e `t2` são termos, existe um termo `u` tal que `parallel_reduction?(E)(t1,u) & parallel_reduction?(E)(t2,u)`. Daí, para provar que, de fato, a redução em paralelo acontece, devemos fornecer sequências de posições, de regras e de substituições tais como na seção 2.10.

O teorema 1.2.4 mostra qual a estrutura desse termo `u`. Ou seja, ao final, precisamos apresentar um termo em que todos os triângulos coloridos vazados foram trocados por triângulos preenchidos.

As sequências `fsp!1`, `fse!1` e `fss!1` de posições, regras e substituições que aparecem a seguir são referentes à redução de `s` a `t1`. Da mesma forma, `fsp!2`, `fse!2` e `fss!2` são as sequências da redução de `s` a `t2`. O termo a seguir é o que nós buscamos.

```
replace_par_pos(t1, Pos_Over(fsp!2,fsp!1) o
      Compo_pos_var(left_rewrite_rules(choose_seq(Pos_Over(fsp!1,fsp!2),fsp!1,fse!1)),
                  right_rewrite_rules(choose_seq(Pos_Over(fsp!1,fsp!2),fsp!1,fse!1)),
                  SEQ_VAR_multi(E,s,Pos_Over(fsp!1,fsp!2),
                                choose_seq(Pos_Over(fsp!1,fsp!2),fsp!1,fss!1),
                                choose_seq(Pos_Over(fsp!1,fsp!2),fsp!1,fse!1),
                                fsp!2,fse!2,fss!2),
                  Pos_Over(fsp!1,fsp!2),fsp!2),
      choose_seq(Pos_Over(fsp!2,fsp!1),fsp!2,fse!2)
        o Compo_rr_var(left_rewrite_rules(choose_seq(Pos_Over(fsp!1,fsp!2),
                                                     fsp!1,fse!1)),
                       right_rewrite_rules(choose_seq(Pos_Over(fsp!1,fsp!2),
                                                      fsp!1,fse!1)),
                       SEQ_VAR_multi(E,s,Pos_Over(fsp!1,fsp!2),
                             choose_seq(Pos_Over(fsp!1,fsp!2),fsp!1,fss!1),
                             choose_seq(Pos_Over(fsp!1,fsp!2),fsp!1,fse!1),
                             fsp!2,fse!2,fss!2),
                       Pos_Over(fsp!1,fsp!2),fsp!2,fse!2),
      SIGMA_multi(Pos_Over(fsp!2,fsp!1),
              choose_seq(Pos_Over(fsp!2,fsp!1),fsp!2,fss!2),
              SEQ_VAR_multi(E,s,Pos_Over(fsp!2,fsp!1),
                            choose_seq(Pos_Over(fsp!2,fsp!1),fsp!2,fss!2),
                            choose_seq(Pos_Over(fsp!2,fsp!1),fsp!2,fse!2),
                            fsp!1,fse!1,fss!1),
              choose_seq(Pos_Over(fsp!2,fsp!1),fsp!2,fse!2),
              Subs_right_rr(fse!1,fss!1),fsp!1)
        o Compo_Sub_var(left_rewrite_rules(choose_seq(Pos_Over(fsp!1,fsp!2),
                                                      fsp!1,fse!1)),
                        right_rewrite_rules(choose_seq(Pos_Over(fsp!1,fsp!2),
                                                       fsp!1,fse!1)),
                        SEQ_VAR_multi(E,s,Pos_Over(fsp!1,fsp!2),
```

```
                        choose_seq(Pos_Over(fsp!1,fsp!2),fsp!1,fss!1),
                        choose_seq(Pos_Over(fsp!1,fsp!2),fsp!1,fse!1),
                        fsp!2,fse!2,fss!2),
                Pos_Over(fsp!1,fsp!2),fsp!2,fss!2))
```

O termo é construído a partir da aplicação de `replace_par_pos` sobre o termo `t1` e as três sequências de posições, de regras e de substituições.

Ao instanciarmos a prova com o termo acima algumas propriedades sobre essas sequências devem ser demonstradas para que fechemos o resultado. Os ramos de prova são automaticamente gerados pelo PVS ao fazer a instanciação. Algumas dessas propriedades já foram demonstradas em outros lemas e outras permanecem em aberto em conjecturas separadas. Dos resultados em aberto, quase todos seguem o tipo de prova realizada em resultados já fechados. Apenas o último resultado da Seção 3.3.2 deverá ter um tratamento diferenciado e também se desdobrará em alguns outros lemas.

3.3.1 Resultados demonstrados

Em relação à aplicação `choose_seq`, nós precisamos provar a conservação do comprimento das sequências e a preservação do índice. Abaixo nós temos alguns resultados, mas podemos considerar também as propriedades espelhadas, ou seja, trocar `fsp!1` por `fsp!2`, `fse!1` por `fse!2` e `fss!1` por `fss!2` e vice-versa. Precisamos considerar também que `Pos_Over` seleciona uma subsequência.

```
- Pos_Over(fsp!1, fsp!2)`length =
        choose_seq(Pos_Over(fsp!1, fsp!2), fsp!1, fse!1)`length
- FORALL (i:below[choose_seq(Pos_Over(fsp!2, fsp!1), fsp!2, fse!2)`length]):
                        E(choose_seq(Pos_Over(fsp!2, fsp!1), fsp!2, fse!2)`seq(i))
- FORALL (i:below[choose_seq(Pos_Over(fsp!2, fsp!1), fsp!2, fse!2)`length]):
                  subtermOF(x, Pos_Over(fsp!2, fsp!1)`seq(i)) =
                    ext(choose_seq(Pos_Over(fsp!2, fsp!1), fsp!2, fss!2)`seq(i))
                       (lhs(choose_seq(Pos_Over(fsp!2, fsp!1), fsp!2, fse!2)`seq(i)))
- FORALL (i: below[Pos_Over(fsp!1, fsp!2)`length]):
        mem_seq(Pos_Over(fsp!1, fsp!2)`seq(i), fsp!1)
```

As aplicações `SEQ_VAR_multi`, `SIGMA_multi`, `left_rewrite_rules` e `Subs_right_rr` também preservam comprimento.

```
- Pos_Over(fsp!1, fsp!2)`length
           = SEQ_VAR_multi(E, s, Pos_Over(fsp!1, fsp!2),
                           choose_seq(Pos_Over(fsp!1, fsp!2), fsp!1, fss!1),
```

```
                          choose_seq(Pos_Over(fsp!1, fsp!2), fsp!1, fse!1),
                          fsp!2, fse!2, fss!2)‘length
- Pos_Over(fsp!2, fsp!1)‘length
          = SIGMA_multi(Pos_Over(fsp!2, fsp!1),
                        choose_seq(Pos_Over(fsp!2, fsp!1), fsp!2, fss!2),
                        SEQ_VAR_multi(E, s, Pos_Over(fsp!2, fsp!1),
                                      choose_seq(Pos_Over(fsp!2, fsp!1),fsp!2,fss!2),
                                      choose_seq(Pos_Over(fsp!2, fsp!1),fsp!2,fse!2),
                                      fsp!1, fse!1, fss!1),
                        choose_seq(Pos_Over(fsp!2, fsp!1), fsp!2, fse!2),
                        Subs_right_rr(fse!1, fss!1), fsp!1)‘length
- Pos_Over(fsp!1, fsp!2)‘length =
      left_rewrite_rules(choose_seq(Pos_Over(fsp!1, fsp!2), fsp!1, fse!1))‘length
- fsp!2‘length = Subs_right_rr(fse!2, fss!2)‘length
```

Já o comprimento de `Compo_pos_var` é preservado por `Compo_rr_var` e `Compo_Sub_var`.

```
- Compo_pos_var(left_rewrite_rules(choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1)),
                right_rewrite_rules(choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1)),
                SEQ_VAR_multi(E, s, Pos_Over(fsp!1, fsp!2),
                           choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fss!1),
                           choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1),
                           fsp!2, fse!2, fss!2),
                Pos_Over(fsp!1, fsp!2), fsp!2)‘length  =
  Compo_rr_var(left_rewrite_rules(choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1)),
               right_rewrite_rules(choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1)),
               SEQ_VAR_multi(E, s, Pos_Over(fsp!1, fsp!2),
                           choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fss!1),
                           choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1),
                           fsp!2, fse!2, fss!2),
               Pos_Over(fsp!1, fsp!2), fsp!2, fse!2)‘length  =
  Compo_Sub_var(left_rewrite_rules(choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1)),
               right_rewrite_rules(choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1)),
               SEQ_VAR_multi(E, s, Pos_Over(fsp!1, fsp!2),
                           choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fss!1),
                           choose_seq(Pos_Over(fsp!1, fsp!2),fsp!1,fse!1),
                           fsp!2, fse!2, fss!2),
               Pos_Over(fsp!1, fsp!2), fsp!2, fss!2)‘length
```

Temos que `Pos_seq_var` conserva o comprimento de `SEQ_VAR_multi`. Isso quer dizer muito, pois temos de usar a linearidade da regra da posição em `Pos_Over(fsp1,fsp2)` e a não ambiguidade do sistema `E`, provando que `set_var` (presente na definição de `SEQ_VAR_multi`) é um conjunto unitário.

```
- Pos_seq_var(lhs(choose_seq(Pos_Over(fsp1, fsp2), fsp1, fse1)`seq(i)),
              SEQ_VAR_multi(E, s, Pos_Over(fsp1, fsp2),
                      choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                      choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                      fsp2, fse2, fss2)`seq(i))`length =
         (SEQ_VAR_multi(E, s, Pos_Over(fsp1, fsp2),
                  choose_seq(Pos_Over(fsp1, fsp2), fsp1, fss1),
                  choose_seq(Pos_Over(fsp1, fsp2), fsp1, fse1),
                  fsp2, fse2, fss2)`seq(i))`length
```

Sabemos que as variáveis que ocorrem em uma determinada sequência dentro de `SEQ_VAR_multi` são todas diferentes. Isso é extremamente importante considerando a definição de `SIGMA_multi` na seção 2.15, que exige essa não repetição de variáveis na sequência.

```
- (SEQ_VAR_multi(E, s, Pos_Over(fsp1, fsp2),
                 choose_seq(Pos_Over(fsp1, fsp2), fsp1, fss1),
                 choose_seq(Pos_Over(fsp1, fsp2), fsp1, fse1),
                 fsp2, fse2, fss2)`seq(i))`seq(m) /=
     (SEQ_VAR_multi(E, s, Pos_Over(fsp1, fsp2),
                    choose_seq(Pos_Over(fsp1, fsp2), fsp1, fss1),
                    choose_seq(Pos_Over(fsp1, fsp2), fsp1, fse1),
                    fsp2, fse2, fss2)`seq(i))`seq(n)
```

Também destacamos que a função `replace_par_pos`, que realiza a redução em paralelo propriamente dita, possui uma independência na ordem em que se tomam as posições a serem reduzidas. Pela definição, ela toma primeiramente a posição com índice 0. No entanto, como podemos ver no resultado seguinte, caso `fsp` seja uma sequência de posições paralelas, podemos tomar um outro índice `n` qualquer da sequência, continuar reduzindo nas posições restantes e ainda assim obter o mesmo termo final.

```
- replace_par_pos(replaceTerm(s, ext(fss(n))(rhs(fse(n))), fsp(n)),
                  delete(fsp,n), delete(fse,n), delete(fss,n))
            = replace_par_pos(s,fsp,fse,fss)
```

Esse último lema é extremamente importante considerando que tomamos posições específicas ao construir o termo de juntabilidade para o problema.

3.3.2 Resultados axiomatizados

O primeiro resultado em aberto é que, nas posições em que `fsp!2` está por cima de `fsp!1`, o subtermo de `t1` é a mesma regra de `fse!2` que se tinha originalmente instanciada com a substituição atualizada por `SIGMAP`.

```
- subtermOF(t1, Pos_Over(fsp2,fsp1)‘seq(i)) =
    ext(SIGMAP(choose_seq(Pos_Over(fsp2,fsp1),fsp2,fss2)‘seq(i),
           seq_var_par(E, s, Pos_Over(fsp2,fsp1)‘seq(i),
                      choose_seq(Pos_Over(fsp2,fsp1),fsp2,fss2)‘seq(i),
                      choose_seq(Pos_Over(fsp2,fsp1),fsp2,fse2)‘seq(i),
                      sub_pos(fsp1,Pos_Over(fsp2,fsp1)‘seq(i)),
                      choose_seq(sub_pos(fsp1,Pos_Over(fsp2,fsp1)‘seq(i)),fsp1,fse1),
                      choose_seq(sub_pos(fsp1,Pos_Over(fsp2,fsp1)‘seq(i)),fsp1,fss1)),
           Pos_seq_var(lhs(choose_seq(Pos_Over(fsp2,fsp1),fsp2,fse2)‘seq(i)),
                      seq_var_par(E,s,Pos_Over(fsp2,fsp1)‘seq(i),
                             choose_seq(Pos_Over(fsp2,fsp1),fsp2,fss2)‘seq(i),
                             choose_seq(Pos_Over(fsp2,fsp1),fsp2,fse2)‘seq(i),
                             sub_pos(fsp1,Pos_Over(fsp2,fsp1)‘seq(i)),
                             choose_seq(sub_pos(fsp1,Pos_Over(fsp2,fsp1)‘seq(i)),
                                       fsp1,fse1),
                             choose_seq(sub_pos(fsp1,Pos_Over(fsp2,fsp1)‘seq(i)),
                                       fsp1,fss1))),
           choose_seq(sub_pos(fsp1,Pos_Over(fsp2,fsp1)‘seq(i)),
                      fsp1,Subs_right_rr(fse1, fss1)),
           complement_pos(Pos_Over(fsp2,fsp1)‘seq(i), fsp1)))
                    (lhs(choose_seq(Pos_Over(fsp2,fsp1),fsp2,fse2)‘seq(i)))
```

Tomemos agora um subtermo numa posição em que `fsp!1` está por cima de `fsp!2`, ou seja, é prefixo de alguma posição em `fsp!2`. Quando reduzimos `s` para `t1`, sabemos que as regras de `fse!2` instanciadas com `fss!2` que estão por baixo desse termo mudaram de posição devido ao movimento das variáveis durante a reescrita. Vimos que essas novas posições são descritas por `Compo_pos_var` e as regras e substituições que a acompanham por `Compo_rr_var` e `Compo_Sub_var` respectivamente. Precisamos provar que, de fato, essas duas últimas preservam o índice da primeira, mostrando que os subtermos de `t1` nas posições de `Compo_pos_var` são o lado esquerdo das regras em `Compo_rr_var` instanciadas com `Compo_Sub_var`. Também devemos provar que as regras em `Compo_rr_var` continuam sendo as regras do sistema ortogonal `E`.

```
- subtermOF(t1,Compo_pos_var(left_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),
                                                          fsp1,fse1)),
```

```
                        right_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),
                                                       fsp1,fse1)),
                        SEQ_VAR_multi(E,s,Pos_Over(fsp1, fsp2),
                            choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                            choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                            fsp2,fse2,fss2),
                        Pos_Over(fsp1, fsp2), fsp2)`seq(i)) =
  ext(Compo_Sub_var(left_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                    right_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                    SEQ_VAR_multi(E,s,Pos_Over(fsp1, fsp2),
                         choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                         choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                         fsp2,fse2,fss2),
                    Pos_Over(fsp1, fsp2), fsp2, fss2)`seq(i))
          (lhs(Compo_rr_var(left_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),
                                                          fsp1,fse1)),
                            right_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),
                                                           fsp1,fse1)),
                            SEQ_VAR_multi(E,s,Pos_Over(fsp1, fsp2),
                              choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                              choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                              fsp2,fse2,fss2),
                            Pos_Over(fsp1, fsp2), fsp2, fse2)`seq(i)))
- E(Compo_rr_var(left_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                 right_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                 SEQ_VAR_multi(E, s, Pos_Over(fsp1,fsp2),
                         choose_seq(Pos_Over(fsp1,fsp2),fsp1,fss1),
                         choose_seq(Pos_Over(fsp1,fsp2),fsp1,fse1),
                         fsp2,fse2,fss2),
                 Pos_Over(fsp1,fsp2),fsp2,fse2)`seq(i))
```

A relação de paralelismo é mantida por `Compo_pos_var`. Porém, mais do que isso, a composição desta sequência com `Pos_Over(fsp2, fsp1)` permanece sendo de posições paralelas do termo `t1`.

```
- SPP?(t1)(Pos_Over(fsp2, fsp1) o
        Compo_pos_var(left_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                      right_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                      SEQ_VAR_multi(E,s,Pos_Over(fsp1, fsp2),
                            choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                            choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                            fsp2,fse2,fss2),
                      Pos_Over(fsp1, fsp2), fsp2))
```

Outro resultado importante é provar que o construtor `SIGMAP` de fato define uma substituição. Observando a ação dele é fácil ver que a sua ação se restringe a variáveis, pois ele toma como base uma outra substituição e modifica a ação sobre um número finito de variáveis específicas. Como podemos ver na seção 2.6, uma substituição deve ser uma aplicação que leva variáveis em termos e que tem o domínio finito. Assim, a afirmação acima justifica esse resultado. Abaixo, nós colocamos `sigma`, `fsv`, `fsp1`, `fst` e `fsp2` como argumentos arbitrários que satisfazem as condições na seção 2.15.

```
- Sub?(SIGMAP(sigma,fsv,fsp1,fst,fsp2))
```

Finalmente, precisamos provar que essas sequências de posições, regras e substituições realmente levam `t1` e `t2` ao mesmo termo. Observemos abaixo que as sequências responsáveis por reescrever `t2` são um espelho das sequências que reduzem `t1`, isto é, trocamos simplesmente `fsp1` por `fsp2`, `fse1` por `fse2`, `fss1` por `fss2` e vice-versa, como falamos anteriormente.

```
- replace_par_pos(t1,
      Pos_Over(fsp2, fsp1) o
         Compo_pos_var(left_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                       right_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                       SEQ_VAR_multi(E,s,Pos_Over(fsp1, fsp2),
                              choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                              choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                              fsp2,fse2,fss2),
                       Pos_Over(fsp1, fsp2),fsp2),
      choose_seq(Pos_Over(fsp2, fsp1), fsp2, fse2) o
         Compo_rr_var(left_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                      right_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                      SEQ_VAR_multi(E,s,Pos_Over(fsp1, fsp2),
                             choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                             choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                             fsp2,fse2,fss2),
                      Pos_Over(fsp1, fsp2),fsp2,fse2),
      SIGMA_multi(Pos_Over(fsp2, fsp1),choose_seq(Pos_Over(fsp2, fsp1),fsp2,fss2),
                       SEQ_VAR_multi(E,s,Pos_Over(fsp2, fsp1),
                                choose_seq(Pos_Over(fsp2, fsp1),fsp2,fss2),
                                choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2),
                                fsp1,fse1,fss1),
                       choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2),
                       Subs_right_rr(fse1, fss1), fsp1) o
```

```
  Compo_Sub_var(left_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                right_rewrite_rules(choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1)),
                SEQ_VAR_multi(E,s,Pos_Over(fsp1, fsp2),
                       choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                       choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                       fsp2,fse2,fss2),
                Pos_Over(fsp1, fsp2),fsp2,fss2))   =
replace_par_pos(t2, Pos_Over(fsp1, fsp2) o
  Compo_pos_var(left_rewrite_rules(choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2)),
                right_rewrite_rules(choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2)),
                SEQ_VAR_multi(E,s,Pos_Over(fsp2, fsp1),
                        choose_seq(Pos_Over(fsp2, fsp1),fsp2,fss2),
                        choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2),
                        fsp1,fse1,fss1),
                Pos_Over(fsp2, fsp1),fsp1),
    choose_seq(Pos_Over(fsp1, fsp2), fsp1, fse1) o
  Compo_rr_var(left_rewrite_rules(choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2)),
               right_rewrite_rules(choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2)),
               SEQ_VAR_multi(E,s,Pos_Over(fsp2, fsp1),
                       choose_seq(Pos_Over(fsp2, fsp1),fsp2,fss2),
                       choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2),
                       fsp1,fse1,fss1),
               Pos_Over(fsp2, fsp1),fsp1,fse1),
    SIGMA_multi(Pos_Over(fsp1, fsp2),choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                SEQ_VAR_multi(E,s,Pos_Over(fsp1, fsp2),
                        choose_seq(Pos_Over(fsp1, fsp2),fsp1,fss1),
                        choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                        fsp2,fse2,fss2),
                choose_seq(Pos_Over(fsp1, fsp2),fsp1,fse1),
                Subs_right_rr(fse2, fss2), fsp2) o
  Compo_Sub_var(left_rewrite_rules(choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2)),
                right_rewrite_rules(choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2)),
                SEQ_VAR_multi(E,s,Pos_Over(fsp2, fsp1),
                        choose_seq(Pos_Over(fsp2, fsp1),fsp2,fss2),
                        choose_seq(Pos_Over(fsp2, fsp1),fsp2,fse2),
                        fsp1,fse1,fss1),
                Pos_Over(fsp2, fsp1),fsp1,fss1))
```

O último resultado da seção anterior será fundamental na formalização desta igualdade, por ditar a independência da ordem das posições a serem reduzidas por `replace_par_pos`, desde que sejam paralelas, o que é o nosso caso.

Capítulo

4

Conclusão

4.1 Considerações Finais

Durante o desenvolvimento do presente trabalho, foi proposta a teoria `orthogonality` como uma expansão para a teoria `trs`, sendo esta última apresentada em [2, 6, 7] e disponível em [1]. Pode-se observar os passos de especificação e formalização desenvolvidos com o objetivo de provar o teorema de que TRS's ortogonais têm a propriedade de confluência.

No trabalho apresentado em [10], o critério de ortogonalidade fraca é utilizado para garatir confluência aplicando a ferramenta de certificação *Certified Termination Analysis* (CeTA). Este critério diz respeito à ortogonalidade incluindo a possibilidade de pares críticos triviais, ou seja, há a sobreposição de regras, mas elas divergem para o mesmo termo. Embora este resultado implique na confluência de sistemas ortogonais, o que foi apresentado não se trata de formalização, mas de certificação. Até onde se sabe não há trabalhos apresentados que mencionem o objetivo de formalizar tal teorema em um assistente de prova de ordem superior, o que é o nosso caso.

A especificação e formalização construídas seguem a ideia proposta na prova analítica, o que torna a prova bastante diagramática. A estratégia de prova foi adotada visando reduzir o problema da existência de um termo de juntabilidade para o Teorema 1.2.4 simplesmente para a verificação de propriedades sobre as funções recursivas criadas na Seção 2.2, o que gerou uma série de resultados cujas provas são indutivas, sendo a maior parte já formalizada e algums lemas ainda permanecem axiomatizados, como mencionado no final da seção anterior.

De fato, trabalhar com teoremas de existência indutivamente pode tornar-se extremamente complicado uma vez que a hipótese de indução nos dá um termo de juntabilidade cujas peculiaridades de comprimento das sequências necessárias (posições, regras e substituições) e da localização das posições reduzidas são desconhecidas e não

conseguimos proceder ao passo indutivo por esse caminho diretamente. Por isso, optamos por descrever exatamente o termo de juntabilidade do lema `parallel_reduction_is_DP` conforme apresentado na Seção 3.3. Não queremos dizer que a abordagem indutiva diretamente dentro dessa prova principal não seja uma opção, mas exige um estudo diferenciado para que o passo indutivo possa ser efetivado.

Pôde-se experimentar várias estratégias de prova como, por exemplo, por indução sobre a medida de sequências, por absurdo, definicionais e outras. Obteve-se a construção de vários conceitos não triviais, além da formalização de lemas sobre eles. Houve também a boa integração entre a teoria principal `orthogonality` e as subteorias com tipos polimórficos que nos dão as ferramentas `chose_seq`, `power`, `seq2set` e `mem_seq`, amplamente utilizadas. Isso pode ser afirmado uma vez que a grande maioria das definições que utilizam os conceitos das subteorias importadas tiveram as obrigações de prova geradas pelo PVS formalizadas, excluindo unicamente a prova de que o construtor `SIGMAP` da Seção 2.15 gera, de fato, uma substituição, o que já discutimos ser verdade quando estudamos o domínio da função gerada.

Um enfraquecimento do teorema principal foi realizado acrescentando a hipótese de linearidade à direita, o que significa que não ambiguidade e linearidade (tanto à direita quanto à esquerda) geram um TRS confluente. A prova fora mais simples e foi um bom objeto de estudo para nos familiarizarmos com a demonstração do teorema de confluência dos sistemas ortogonais. Este resultado fora formalizado completamente em 2010 e foi apresentado em [9].

Atualmente o desenvolvimento completo consiste em 40 definições, 84 lemas formalizados e 8 axiomas distribuídos em cerca 1300 linhas de especificação e, adicionalmente, 46000 linhas de provas. Dos 8 axiomas assumidos, 7 devem ser lemas fechados com uma relativa facilidade devido à semelhança semântica com outros resultados já demonstrados. A prova do último resultado apresentado na Seção 3, que trata da equivalência da ação da função `replace_par_pos` através de dois conjuntos de argumentos específicos, é o nosso objetivo mais desafiador e ainda deve se desdobrar em alguns outros lemas intermediários.

O PVS, sendo um assistente de prova com lógica de ordem superior, foi a ferramenta escolhida por tratar esses resultados sobre relações de reescrita quaisquer de um modo natural, lidando assim com objetos de segunda ordem. Isso implica consequentemente que o PVS dá um bom suporte para tratar corretude, completude e restrições de integridade das especificações de objetos computacionais especificados através de regras de reescrita.

Dos lemas formalizados, destacamos três resultados importantes dentro da teoria `orthogonality`:

- primeiramente, provamos a equivalência entre o fecho reflexivo transitivo da reescrita usual e o da reescrita em paralelo. Isso foi realizado provando a inclusão da

reescrita usual na reescrita em paralelo, o que é trivial, e da reescrita em paralelo no fecho reflexivo transitivo da reescrita usual, fazendo uma prova indutiva sobre o comprimento da sequência de posições responsável pela reescrita em paralelo.

- demonstrou-se também a propriedade do diamante para a reescrita em paralelo num TRSs ortogonal. Esse resultado necessitou de uma série de lemas auxiliares técnicos para mostrar que o termo de juntabilidade fornecido atende às condições necessárias. Todo o ferramental utilizado está descrito na Seção 2.2, e as principais condições exigidas na prova estão na Seção 3.3. Alguns dos lemas auxiliares foram axiomatizados. Citamos, por exemplo, que necessita-se mostrar que as aplicações `compo_pos_var`, `compo_rr_var` e `compo_Sub_var` preservam ordem entre si. No entanto, a maior parte dos lemas foi formalizada como, por exemplo, a preservação de comprimento e ordem por `choose_seq` e a independência da ordem de redução por `replace_par_pos`.

- por último, reuniram-se dois lemas da teoria `ars` aos dois itens anteriores para mostrar que ortogonalidade implica em confluência. Sabe-se que a propriedade do diamante acarreta na propriedade Church-Rosser, que é equivalente à confluência. Assim, mostramos que a reescrita em paralelo é confluente a partir do item anterior e isso é igual à confluência da reescrita usual, pelo primeiro item.

4.2 Sugestões para Pesquisas Futuras

Para que o teorema de confluência para TRS's ortogonais seja, de fato, certificado, é necessário apresentar as demonstrações para os resultados axiomatizados. Assim, sugere-se como trabalho futuro:

- a conclusão da formalização da forma como foi apresentada (demonstração dos lemas axiomatizados);

- o estudo de um caminho alternativo para tratar a prova de `parallel_reduction_is_DP` de forma indutiva diretamente;

- a formalização de outras condições para confluência e/ou terminação no contexto de TRS's;

- partir para outros contextos computacionais importantes como a Teoria de Tipos e realizar um trabalho semelhante à das teorias `trs` e `ars`, que formam um arcabouço robusto para a formalização de teoremas relevantes.

Referências Bibliográficas

[1] *Theory* `trs`, (consulted January 2012): *Available in the NASA LaRC PVS library*, http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html. 1, 19, 51, 60

[2] A. B. Avelar, F. L. C. de Moura, A. L. Galdino, and M. Ayala-Rincón, *Verification of the completeness of unification algorithms à la robinson*, WoLLIC 2010, 2010, pp. 110–124. 1, 19, 60

[3] Franz Baader and Tobias Nipkow, *Term rewriting and All That*, Cambridge University Press, 1998. 2, 13

[4] M. Bezem, J.W. Klop, and R. de Vrijer, *Term rewriting systems by TeReSe*, Cambridge Tracts in Theoretical Computer Science, no. 55, Cambridge University Press, 2003. 2, 13

[5] A. L. Galdino and M. Ayala-Rincón, *A formalization of newman's and yokouchi lemmas in a higher-order language*, Journal of Automated Reasonning **1** (2008), no. 1, 39–50. 1, 2

[6] ———, *A PVS theory for term rewriting systems*, Electronic Notes in Theoretical Computer Science **247** (2009), 67–83, Third Workshop on Logical and Semantic Frameworks, with Applications - LSFA 2008. 1, 2, 19, 27, 60

[7] ———, *A formalization of the Knut-Bendix(-Huet) critical pair theorem*, Journal of Automated Reasonning **45** (2010), no. 3, 301–325. 1, 19, 27, 60

[8] A.L. Galdino and M. Ayala-Rincón, *A Theory for Abstract Rewriting Systems in PVS*, Proc. XXXIII CLEI, 2007, pp. 1–16. 2, 27

[9] A. C. R. Oliveira and M. Ayala-Rincón, *On the formalization of confluence of orthogonal rewriting systems in PVS*, EBL 2011, 2011, p. 15. 1, 2, 61

[10] R. Thiemann, *Certification of confluence proofs using CeTA*, First International Workshop on Confluence (IWC 2012), 2012, p. 45. 60