Universidade Federal Fluminense

FABRICIO CHALUB BARBOSA DO ROSÁRIO

Uma Implementação de Semântica Operacional Estrutural Modular em Maude

NITERÓI

2005

Universidade Federal Fluminense

FABRICIO CHALUB BARBOSA DO ROSÁRIO

Uma Implementação de Semântica Operacional Estrutural Modular em Maude

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração:Processamento Distribuído e Paralelo/Métodos Formais.

Orientador:
Christiano de Oliveira Braga

NITERÓI

2005

Uma Implementação de Semântica Operacional Estrutural Modular em Maude

Fabricio Chalub Barbosa do Rosário

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Banca examinadora:

Prof. Christiano de Oliveira Braga, D. Sc. / IC-UFF
(Orientador)

Prof. Edward Hermann Haeusler, D. Sc. / PUC-Rio

Prof. Peter D. Mosses, Ph. D. / Univ. of Wales Swansea

Niterói, 27 de maio de 2005.

a meus pais, meus grandes incentivadores, e à minha querida irmã.

Agradecimentos

A Christiano de Oliveira Braga pela competente orientação e zelo durante estes dois anos e meio de trabalho e pelas oportunidades que me ofereceu durante o curso.

Ao professor Peter D. Mosses pelo caloroso acolhimento em Aarhus, Dinamarca, durante o mês de novembro de 2004.

A Jørgen Iversen, Janus Dam Nielsen, Claus Braband, Marco Carbone e Karen Kjær Møller pela ajuda e amizade.

À Fundação Getúlio Vargas, Escola de Pós-Gradução em Economia, nas pessoas de Alexandre Rademaker e dos professores Clóvis de Faro e Renato Fragelli pela oportunidade que me proporcionaram de realizar meu mestrado na Universidade Federal Fluminense.

Ao CNPq e PROPP/UFF pela bolsa a mim concedida para aprofundar meus conhecimentos em Aarhus, Dinamarca.

A meus amigos Alexandre Furtado, Alexandre Rademaker Alexandre Rocha, Ana Paula, Carlos Jr., Christiano Braga, Guilheme e Amanda (e a Manuzinha), Pedro, Rodrigo Rocha, Rodrigo Taranto e Sergio.

Resumo

Esta dissertação apresenta uma ferramenta formal para Semântica Estrutural Operacional Modular (MSOS, na sigla em inglês), utilizando-se da conversão de MSOS para Lógica de Reescrita recentemente desenvolvida por Braga e Meseguer. A implementação, denominada Maude MSOS Tool (MMT), foi desenvolvida em Maude, uma implementação de Lógica de Reescrita de alta performance. O desenvolvimento de MMT objetiva não somente desenvolver uma ferramenta que utiliza uma linguagem de especificação que está mais próxima do domínio MSOS do que da Lógica de Reescrita, como também demonstrar o que ganhamos ao desenvolver um ambiente completamente formal em Maude, uma vez que isto possibilita o uso de outras ferramentas formais disponíveis para especificações MSDF. Demonstramos isto através de simulação e verificação de algoritmos distribuídos e programas concorrentes. Outro objetivo é exemplificar uma extensão não-trivial de Full Maude.

Abstract

This dissertation presents a formal tool for Modular Structural Operational Semantics (MSOS), based on the conversion from MSOS to Rewriting Logic recently developed by Braga and Meseguer. The implementation, named *Maude MSOS Tool* (MMT), was written in Maude, a high-performance implementation of Rewriting Logic. The development of MMT attempts not only to provide an MSOS interpreter that uses a specification language that is closer to the domain of MSOS specifications than to Maude specifications, but also to demonstrate what can be accomplished when one develops a formal tool in the Maude environment, since it allows the use of other formal tools already available with MSDF specifications. We have demonstrated this by simulating and model checking concurrent programs and distributed algorithms. Another aim is to provide an example of a non-trivial extension of Full Maude and to create a tool that is itself extensible.

Palavras-chave

1. semântica de linguagens de programação
2. semântica operacional estrutural modular
3. lógica de reescrita

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Semântica Operacional Estrutural (*Structural Operational Semantics*, SOS), desenvolvido por Plotkin [1], é um *framework* comumente usado na especificação formal de linguagens de programação [2, 3] e sistemas concorrentes [4, 5]. É também amplamente usado em livros-texto sobre semântica formal e notas de aula [6, 7, 8, 9, 1, 10]. Infelizmente, se olharmos sob a óptica da engenharia de software, falta à SOS uma característica fundamental para a especificação de sistemas complexos: *modularidade*. Mosses resolveu este problema com o desenvolvimento de SOS Modular (*Modular SOS*, MSOS) [11]. Recentemente, Mosses também desenvolveu uma linguagem de especificação para MSOS, o Formalismo de Especificação para SOS Modular (*Modular SOS Specification Formalism*, MSDF) [10].

Uma vasta quantidade de métodos algébricos foram desenvolvidos nos últimos anos, especialmente no contexto de formalização da semântica de linguagens de programação. Lógica de Reescrita (*Rewriting Logic*) [12] e Lógica Equacional de Pertinência (*Membership Equational Logic*) [13] são dois notáveis exemplos que foram usados para a especificação de uma grande variedade de tópicos [14, 15, 16, 17, 18, 19, 20, 21, 22].

A relação entre Lógica de Reescrita e SOS [23, 18, 24, 20, 25] e, em particular MSOS [25, 20] já é bem conhecida e estudada.

Esta dissertação almeja fechar este círculo provendo um ambiente formal para especificações MSDF, utilizando-se da conversão de MSOS para Lógica de Reescrita existente. A implementação, denominada *Maude MSOS Tool* (MMT) foi desenvolvida em Maude [26], uma implementação de Lógica de Reescrita de alta-performance. O desenvolvimento de MMT objetiva não somente desenvolver uma ferramenta que utiliza uma linguagem de especificação que está mais próxima do domínio MSOS do que da Lógica de Reescrita, como também demonstrar o que ganhamos ao desenvolver um ambiente completamente formal

em Maude, uma vez que isto possibilita o uso de outras ferramentas formais disponíveis para especificações MSDF. Demonstramos isto através de simulação e verificação de algoritmos distribuídos e programas concorrentes. Outro objetivo é exemplificar uma extensão não-trivial de Full Maude, além de criar uma ferramenta que é ela própria extensível.

Esta dissertação está organizada da seguinte maneira. O capítulo 2 contém o material básico necessário sobre os *frameworks* necessários para explicar formalmente a transformação de MSOS para Lógica de Reescrita e sua implementação em Maude; o capítulo 3 mostra outras implementações de SOS e MSOS; o capítulo 4 descreve a sintaxe da linguagem MSDF, a linguagem de especificação usada pelo *Maude MSOS Tool*; o capítulo 5 descreve a implementação de *Maude MSOS Tool*; o capítulo 6 mostra diversas aplicações do MMT à especificação e verificação de linguagens de programação e sistemas distribuídos. O capítulo 7 conclui esta dissertação com alguns comentários finais. Os apêndices contêm material adicional que foi omitido de determinadas seções, especialmente do capítulo 6 para tornar a apresentação dos temas principais mais sucinta.

Capítulo 2

Fundamentos

Este capítulo provê o material fundamental sobre Semântica Operacional Estrutural [1] (seção 2.1), Lógica de Reescrita [12] (seção 2.2) e Maude [26] (seção 2.3); a relação entre Semântica Operacional Estrutural Modular (MSOS) e Lógica de Reescrita é feita primeiramente introduzindo a Semântica de Reescrita Modular [25, 20] (seção 2.4). Mostra-se em seguida como formalizar MSOS com Semântica de Reescrita Modular.

2.1 Semântica Operacional Estrutural e Semântica Operacional Estrutural Modular

2.1.1 Semântica Operacional Estrutural

A *Semântica Operacional Estrutural* (SOS), descrita por Plotkin em [1], é um *framework* teórico comumente usado para a definição formal da semântica de linguagens de programação [3] e sistemas concorrentes [4, 5].

A semântica operacional de uma linguagem de programação em SOS é dada por um sistema de transição terminal e rotulado $(\Gamma, A, \rightarrow, T)$, onde Γ é um conjunto (de configurações $\gamma \in \Gamma$), A um conjunto de rótulos, $\rightarrow \subseteq \Gamma \times A \times \Gamma$ uma relação ternária, e $T \subseteq \Gamma$ é o conjunto de configurações terminais tal que se $(\gamma, \alpha, \gamma') \in \rightarrow$ então $\gamma \notin T$. Especificações SOS são pares (S, T), onde S é a definição da sintaxe abstrata e T é o conjunto de transições. Uma transição $t \in T$ é especificada usando a notação $\gamma \xrightarrow{a} \gamma'$ que significa $(\gamma, a, \gamma') \in \rightarrow$, ou seja, existe uma transição da configuração γ para a configuração γ', com rótulo a. Transições condicionais são especificadas da seguinte forma:

$$\frac{c_1, \ldots, c_n}{c}$$

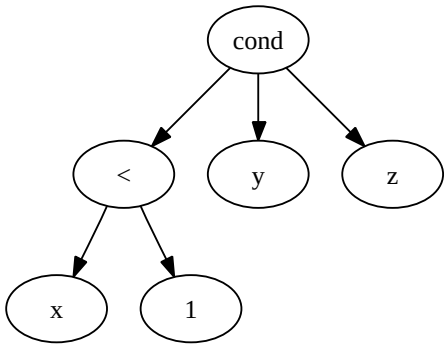

Figura 2.1: A árvore sintática com valores `cond(x < 1, y, z)`

onde a conclusão $\mathtt{c}$ é uma transição, e cada condição $\mathtt{c}_\mathtt{i}$ é ou uma transição ou algum outro tipo de condição tal como equações, pertinência de conjuntos, etc.

As configurações $\gamma \in \Gamma$ são tuplas contendo árvores sintáticas com valores (*value-added syntactic trees*)—ou seja, árvores sintáticas onde alguns galhos podem conter valores finais—e possivelmente componentes semânticos, tais como o ambiente de amarrações (*bindings environment*), armazenagem (*stores*), etc. Árvores sintáticas com valores são também chamadas de "termos"; figura 2.1 exibe a árvore para o termo `cond(x < 1, y, z)`, de agordo com uma gramática hipotética que especifica a sintaxe de uma construção condicional no formato prefixado. O '1' deve ser visto como um valor final computado. O uso de sintaxe abstrata segue a prática tradicional de evitar complicações que surgem da complexidade da sintaxe concreta das linguagens de programação que em nada adicionam ao entendimento preciso da semântica. Construções (abstratas) de uma linguagem têm a sua semântica definida por construções matemáticas, chamadas de *componentes semânticos* em SOS. Por exemplo, o ambiente de amarrações é normalmente modelado como uma função que mapeia identificadores para valores, $\rho : \mathcal{I} \to \mathbb{N}$ tal que escrevemos $\mathtt{m} = \rho(\mathtt{i})$, $\mathtt{i} \in \mathcal{I}$ para acessar o valor $\mathtt{m} \in \mathbb{N}$ amarrado ao identificador $\mathtt{i}$ no ambiente ρ. Outros tipos de funções (ou relações) podem ser definidos como componentes.

Plotkin deixou em aberto o problema da modularidade em especificações SOS. Por exemplo, as regras 2.1 e 2.2 definem a SOS de expressões matemáticas. Primeiro, vamos exibir a sintaxe abstrata.

$$\begin{array}{rcl} \mathtt{m} & \in & \mathbb{N} \\ \mathtt{e} & \in & \mathsf{Exp} \\ & & \mathtt{e} ::= \mathtt{m} \mid \mathtt{e}_0 + \mathtt{e}_1 \end{array}$$

As regras de transição dão o significado da expressão matemática $\mathtt{e}_0 + \mathtt{e}_1$. Inicialmente

a expressão e_0 é computada até que atinja um valor final, um número natural (m_0); o mesmo é feito para a expressão e_1 (m_1). A regra 2.2 reescreve $m_0 + m_1$ para a soma dos naturais de m_0 e m_1.

$$\frac{e_0 \longrightarrow e_0'}{e_0 + e_1 \longrightarrow e_0' + e_1} \qquad \frac{e_1 \longrightarrow e_1'}{m_0 + e_1 \longrightarrow m_0 + e_1'} \tag{2.1}$$

$$m_0 + m_1 \longrightarrow m_0 + m_1 \tag{2.2}$$

A adição de amarrações, por exemplo, através da introdução de uma construção similar à '`let`' de ML requer a adição de um componente para modelar este ambiente ($\rho \in Env$) à configuração. Nesta especificação simples, ambientes são funções finitas de variáveis para valores finais $Var \to \mathbb{N}$.

Mostraremos aqui um exemplo adaptado das notas de aula de Plotkin, que especifica uma forma mais simples da construção '`let`' da linguagem Standard ML [3].

A regra 2.3 especifica que, primeiro, a expressão e_0 deve ser computada até que um valor final m seja encontrado. A regra 2.4 especifica que e_1 deve ser computada para e_1' no contexto de um novo ambiente obtido subsitituindo-se todas as instâncias da variável x no ambiente ρ por m e colocando de volta a expressão computada e_1' no corpo da construção '`let`'. Finalmente, a regra 2.5 especifica que quando e_1 é completamente computada para um valor final n, a expressão completa deve ser substituída por n.

$$e ::= \texttt{let } x\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} \qquad x \in Var = \{x_1, x_2, \ldots\}$$

$$\frac{\rho \vdash e_0 \longrightarrow e_0'}{\rho \vdash \texttt{let } x\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} \longrightarrow \texttt{let } x\texttt{=}e_0' \texttt{ in } e_1 \texttt{ end}} \tag{2.3}$$

$$\frac{\rho[m/x] \vdash_{V \cup \{x\}} e_1 \longrightarrow e_1'}{\rho \vdash \texttt{let } x\texttt{=}m \texttt{ in } e_1 \texttt{ end} \longrightarrow \texttt{let } x\texttt{=}m \texttt{ in } e_1' \texttt{ end}} \tag{2.4}$$

$$\rho \vdash \texttt{let } x\texttt{=}m \texttt{ in } n \texttt{ end} \longrightarrow n \tag{2.5}$$

A regra 2.6 localiza o valor amarrado à variavel x no ambiente ρ.

$$\frac{\rho \vdash m = \rho(x)}{\rho \vdash x \longrightarrow m} \tag{2.6}$$

Como agora as expressões são computadas na presença de um ambiente, as regras para expressões matemáticas devem ser reescritas.

$$\frac{\rho \vdash e_0 \longrightarrow e_0'}{\rho \vdash e_0 + e_1 \longrightarrow e_0' + e_1} \qquad \frac{\rho \vdash e_1 \longrightarrow e_1'}{\rho \vdash m_0 + e_1 \longrightarrow m_0 + e_1'} \tag{2.7}$$

$$\rho \vdash m_0 + m_1 \longrightarrow m_0 + m_1 \tag{2.8}$$

As regras especificadas até agora foram descritas no formato chamado de semântica operacional *small-step*, já que a computação da ávore sintática com valores é feita um passo a cada vez, substituindo galhos da árvore nas transições até que um valor final é (possivelmente) alcançado. Uma alternativa para semântica *small-step* é a chamada semântica *big-step*, em que a ávore sintática é *diretamente* computada para um valor final. A semântica de Standard ML é dada em estilo *big-step*, por exemplo [3]. Opcionalmente, a relação *big-step* pode usar o símbolo $\Rightarrow$ para distinguí-la do estilo *small-step*. Para exemplificar, iremos exibir as regras 2.1 e 2.2 reunidas na regra 2.9 no estilo *big-step*.

$$\frac{e_0 \Rightarrow m_0 \quad e_1 \Rightarrow m_1}{e_0 + e_1 \Rightarrow m_0 + m_1} \tag{2.9}$$

Um conceito adicional em SOS é o de uma *extensão operacional conservativa* [27]. Uma extensão de um conjunto de regras é *operacionalmente convservativo* se as transições prováveis no sistema original são as mesmas no sistema estendido. Em [27] é demonstrado que regras *dependentes da origem* (*source-dependent*) formam uma condição necessária para extensões operacionais conservativas. Uma regra é dependente da origem se todas as suas (meta)variáveis são dependentes da origem. As variáveis dependentes da origem numa regra de transição r são obtidas indutivamente como se segue: (i) todas as variáveis na origem de r são dependentes da origem; (ii) se $t \rightarrow t'$ é uma premissa de r e todas as variáveis em t são dependentes da origem, então todas as variáveis em t' são também dependentes da origem. Para ilustrar este conceito, considere o seguinte exemplo, copiado de [27, 28]. Considere duas constantes a e b e uma metavariável x. Com a seguinte regra apenas, não é possível provar $a \rightarrow a$.

$$\frac{\mathtt{x} \rightarrow \mathtt{x}}{\mathtt{a} \rightarrow \mathtt{a}}$$

Contudo, se estendermos o sistema adicionando a seguinte regra, torna-se possível provar $\mathtt{a} \rightarrow \mathtt{a}$, instanciando-se a metavariável $\mathtt{x}$ com a constante $\mathtt{b}$. O problema é que a metavariável $\mathtt{x}$ não é dependente da origem.

$$\mathtt{b} \rightarrow \mathtt{b}$$

2.1.2 Semântica Operacional Estrutural Modular

Para solucionar o problema da modularidade em SOS, Mosses desenvolveu o *framework* denominado SOS Modular (*Modular SOS*, MSOS) [11].[1] A chave para a modularidade de MSOS está no uso de um *sistema de transição generalizado*, em que os componentes semânticos, como o ambiente, foram movidos das configurações para o rótulo da transição. As configurações nas transições consistem apenas de árvores sintáticas com valores. Os rótulos das transições são vistos agora como *setas* de uma categoria e rótulos adjacentes devem ser componíveis. Componentes semânticos, agora no rótulo, são acessíveis através de *índices*. A idéia é que um rótulo pode conter um número não especificado de componentes, mas *apenas os componentes necessários* numa transição devem ser explicitamente referenciados. Formalmente ([11]) um sistema de transição generalizado é uma quádrupla $(\Gamma, \mathbb{A}, \rightarrow, \mathsf{T})$ onde $\mathbb{A}$ é uma categoria com setas A, tal que $(\Gamma, \mathsf{A}, \rightarrow, \mathsf{T})$ é um sistema de transição terminal rotulado. A computação requer que, sempre que uma transição com rótulo α é seguida por uma com rótulo α', é necessário que α e α' sejam componíveis em $\mathbb{A}$.

Já que rótulos são agora setas de uma categoria, iremos discutir como as *categorias de rótulos* são usadas para modelar a informação processada em transições MSOS. Inicialmente, vamos relembrar rapidamente a definição da composição de uma categoria: (i) um conjunto de objetos O; (ii) com conjunto de setas A; (iii) funções *origem* e *destino* de A para O; uma função parcial de $\mathsf{A} \times \mathsf{A}$ para A para composição de setas; e (iv) uma função de O para A mapeando uma seta de identidade para cada objeto. Rótulos que são setas de identidade modelam transições *não-observáveis*, que serão discutidas nesta seção. Normalmente, três diferentes tipos de categorias de rótulos são usadas em especificações

[1] A descrição de MSOS nesta seção segue a descrição dada por Mosses em [11, 10].

MSOS:

- a *categoria discreta* (*discrete*), que contém uma única seta, a identidade, para cada objeto. Estes rótulos representam informação que pode ser lida por uma transição mas não modificada, como é o caso de *ambientes* (informação *read-only*);
- a formada pelo produto cartesiano $\mathsf{O} \times \mathsf{O}$, com setas $(\mathsf{o}, \mathsf{o}')$ indo de o para o'. Composição nesta categoria elimina objetos intermediários e setas identidade têm o formato (o, o); usualmente usada para modelar informação que pode ser lida e modificada por uma transição, como no caso de uma *memória* (informação *read-write*);
- a categoria com apenas um objeto, em que o conjunto de setas é O^*, o *monóide* de seqüências gerado por O. A seta identidade é a seqüência vazia (ε), e composição de setas é concatenação de seqüências (dada pela operação binária $\cdot$); é usada para modelar informação *emitida* ou *produzida* por uma transição, tal como a sinalização de uma exceção ou a impressão de um valor (informação *write-only*).

As mesmas considerações para extensões operacionais conservativas também são aplicáveis para MSOS com a noção de dependência da origem estendida para metavariáveis que apareçam nos rótulos.

Iremos agora descrever especificações MSOS, que são triplas $(\mathsf{S}, \mathsf{L}, \mathsf{T})$, em que S é, assim como em SOS, a sintaxe abstrata, L é a *composição do rótulo* especificada como um produto das três categorias descritas acima, e T é o conjunto de transições. Iremos prosseguir com o entendimento intuitivo de expressões de rótulos em MSOS, descrito no início desta seção (os aspectos categóricos de MSOS são formalmente descritos em [11]). Em MSOS, como mencionamos, componentes são acessíveis em rótulos através de índices que podem ser de três diferentes tipos, que, em última instância, refletem sua natureza categórica: *read-only*, *read-write*, *write-only*.

MSOS define uma notação para os índices de cada tipo diferente de componente: um único índice, não primalizado, $\mathtt{i}$ está associado a um componente *read-only* que é o mesmo no início e ao final da transição; um par de índices $\mathtt{i}$, $\mathtt{i}'$ está associado a um componente *read-write*: o índice não primalizado refere-se à informação presente no início da transição e o índice primalizado refere-se à informação presente ao final da transição; um único índice, primalizado, $\mathtt{i}'$ está associado com componentes *write-only* e se refere apenas à informação produzida e presente ao final da transição.

Os diferentes tipos de componentes têm diferentes requisitos para a componibilidade de rótulos adjacentes. Dois rótulos (L_1, L_2) são componíveis se e somente se L_1 e L_2 têm o mesmo conjunto de índices e, para cada índice i:

- se i indexa um componente *read-only*, $L_1.i = L_2.i$;
- se i, i' indexam um componente *read-write*, $L_1.l' = L_2.i$.

Componentes *write-only* não afetam a componibilidade de rótulos.

O resultado da composição de rótulos $L_1; L_2$ é determinado pela composição de cada par índice-componente (i, c) (também chamado de *campo—field*) pareados pelos seus respectivos índices, como se segue:

- para índices *read-only* $(i, c); (i, c) = (i, c)$, os componentes são os mesmos;
- para índices *read-write* $(i, c, i', c'); (i, c', i', c'') = (i, c, i', c'')$, ou seja, a composição elimina componentes intermediários;
- para índices *write-only* $(i', c); (i', c') = (i', c \cdot c')$, ou seja, a composição é dada em termos da operação binária da monóide livre gerada por i'.

Rótulos em MSOS podem ser denominados *não-observáveis* quando componentes de *read-write* não mudam, e nenhuma informação é produzida por componentes *write-only*. Ou seja, L é um rótulo não-observável, se e somente se, para cada índice i:

- se i e i' indexam um componente *read-write*, $L.i = L.i'$;
- se i' indexa um componente *write-only*, $L.i' = \varepsilon$.

Uma regra de transição incondicional é escrita da seguinte maneira: $t - \alpha \rightarrow t'$ e especifica a relação ternária entre t, t' e o rótulo α. Transições condicionais são como em SOS:

$$\frac{c_1, \ldots, c_n}{c}$$

e especificam que, se as condições $c_1, \ldots, c_n$ são válidas, então a conclusão c é também válida.

As expressões de rótulos α usam uma notação reminiscente da notação em Standard ML para *record patterns*. Cada campo é escrito como $i = c$, onde i é o índice e c o componente, e o "resto do rótulo" é escrito com a notação '...'. Por exemplo, um rótulo α e um índice ρ tal que $\alpha.\rho = \rho_0$ é escrito como $\{\rho = \rho_0, \ldots\}$; um rótulo α com índices σ e σ' tal que $\alpha.\sigma = \sigma_0$ e $\alpha.\sigma' = \sigma_1$ é escrito como $\{\sigma = \sigma_0, \sigma' = \sigma_1, \ldots\}$; e um rótulo α com um índice τ' tal que $\alpha.\tau' = \tau_0$ é escrito como $\{\tau' = \tau_0, \ldots\}$. A metavariáveil X atua sobre rótulos arbitrários, e a metavariável U sobre rótulos não-observáveis. Opcionalmente, ao invés de escrevermos $t \xrightarrow{U} t'$, podemos escrever simplesmente $t \longrightarrow t'$.

Para ilustrarmos MSOS, iremos revisitar as especificações para expressões aritméticas e expressões '`let`'. As regras 2.10 e 2.11 especificam, em MSOS, a computação de expressões aritméticas.

$$\frac{e_0 \xrightarrow{X} e_0'}{e_0 + e_1 \xrightarrow{X} e_0' + e_1} \qquad \frac{e_1 \xrightarrow{X} e_1'}{m_0 + e_1 \xrightarrow{X} m_0 + e_1'} \tag{2.10}$$

$$m_0 + m_1 \longrightarrow m_0 + m_1 \tag{2.11}$$

Para dar semântica a uma expressão '`let`', nós adicionamos o ambiente à especificação através de uma declaração de índice nos rótulos. As regras 2.12, 2.13 e 2.14 especificam em MSOS o significado da construção '`let`'. A descrição informal da regra 2.13 é: para computar a expressão e_1 dentro do '`let`', compute um passo de e_1 para e_1' no contexto de um novo ambiente indexado por env ($\rho[m/x]$); quaisquer mudanças para componentes *read-write* e qualquer informação produzida por componentes *write-only* (representados pela metanotação '...') deve ser transposta para a conclusão.

$$\frac{e_0 \xrightarrow{X} e_0'}{\texttt{let } x = e_0 \texttt{ in } e_1 \texttt{ end} \xrightarrow{X} \texttt{let } x = e_0' \texttt{ in } e_1 \texttt{ end}} \tag{2.12}$$

$$\frac{e_1 \xrightarrow{\{env = \rho[m/x], \ldots\}} e_1'}{\texttt{let } x = m \texttt{ in } e_1 \texttt{ end} \xrightarrow{\{env = \rho, \ldots\}} \texttt{let } x = m \texttt{ in } e_1' \texttt{ end}} \tag{2.13}$$

$$\texttt{let } x = m \texttt{ in } n \texttt{ end} \xrightarrow{U} n \tag{2.14}$$

Finalmente, a regra 2.15 é análoga à regra 2.6. Estamos usando um rótulo não-observável, representado pela metavariável U, e acessando o ambiente indexado por env usando a notação $U.env$.

$$\frac{n = U.env(x)}{x \xrightarrow{U} n} \tag{2.15}$$

As transições em MSOS podem opcionalmente operar sobre ávores sintáticas com valores *tipadas*. O tipo da ávore na conclusão deve ser o mesmo do que o tipo na origem da transição. Esta variedade de árvore sintática especifica uma nova propriedade que deve ser levada em conta no casamento de padrões durante a escolha de qual regra deve ser aplicada a uma determinada ávore, que é o *tipo* da árvore. Para entender este novo requisitos, observe que, num ambiente de inclusão de conjuntos, um determinado termo pode fazer parte de mais de um conjunto—um exemplo simples é o termo '`100`', que faz parte, digamos, dos conjuntos $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ e $\mathbb{Q}$. Com este requisito adicional, podemos criar regras específicas para *cada tipo* que um termo pode ter. Por exemplo: uma especificação hipotética em que temos componentes distintos para as amarrações de valores e as amarrações de funções, podemos criar regras que especificam que um identificador, quando está sendo computador no contexto de uma '`MathExpression`' (expressão matemática), deve buscar seu valor no ambiente que mapeia identificadores para valores, mas quando computado no contexto de uma '`FunctionCall`' (chamada de função), deve buscar seu valor no ambiente que mapeia identificadores a *closures*, e assim por diante.

2.2 Lógica de Reescrita

Lógica de reescrita (*Rewriting Logic*, RWL) [12] é uma lógica de *mudança* em que aspectos estáticos e dinâmicos de um sistema podem ser especificados. É também um *framework lógico* que pode representar diferentes lógicas, linguagens, formalismos operacionais e modelos de computação [14, 15, 16, 17, 18, 19, 20, 21, 22]. Os aspectos dinâmicos são especificados na lógica de reescrita propriamente, usando *regras de reescritas condicionais e rotuladas* e os aspectos estáticos são especificados usando uma lógica equacional. A lógica de reescrita tem diversas implementações de alta performance [29, 26, 30].

Esta seção descreve formalmente lógica de reescrita e sua sublógica equacional, a Lógica Equacional de Pertinência (*Membership Equational Logic*, MEL) [13]. Exemplos práticos são dados na seção 2.3, onde descrevemos a implementação de RWL de nossa escolha, Maude [26].

A sublógica MEL é uma generalização da lógica ordenada-sortida (*order-sorted logic*) em que cada termo pertence a um *kind* e cada *kind* `k` tem um conjunto parcialmente

ordenado $(S_k, \leq)$ de *sorts*. Este esquema permite a representação de parcialidade através da definição de *termos incorretos* como termos que têm *kinds* mas não *sorts*. Um exemplo detalhado é dado usando a sintaxe de Maude na seção 2.3.

Formalmente,[2] uma assinatura em MEL é uma tripla $\Omega = (K, \Sigma, S)$ em que K é um conjunto de *kinds*, Σ é uma assinatura K-tipada $\{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$, e S é uma família de sorts $S = \{S_k\}_{k\in K}$. Seguindo a notação usual, denotamos com T_Σ a álgebra K-tipada de Σ-termos instanciados (*ground*), e por $T_{\Sigma(X)}$ a álgebra K-tipada de Σ-termos sobre o conjunto K-*kinded* de variáveis X.

As fórmulas atômicas de MEL são *equações* $t = t'$, onde t e t' são Σ-termos do mesmo *kind*, ou *axiomas de pertinência* da forma $t : s$, onde t tem *kind* k e $s \in S_k$. Sentenças em MEL são cláusulas Horn sobre estas formas atômicas:

$$(\forall X)\, A_0 \Leftarrow A_1 \wedge \cdots \wedge A_n$$

em que A_i é uma equação ou axioma de pertinência, e cada $x_j \in X$ é uma variável K-*kinded*. Uma teoria em MEL é um par (Ω, E) em que E é o conjunto de sentenças composto de equações condicionais e axiomas condicionais de pertinência sobre a assinatura Ω.

Dada uma teoria MEL $T = (\Omega, E)$ dizemos que uma sentença φ é dedutível a partir de T, e escrevemos $T \vdash \varphi$, se e somente se φ é obtida pela aplicação finita das seguintes regras de dedução:

- Reflexividade.

$$\frac{}{E \vdash (\forall X)\, t = t}$$

- Simetria.

$$\frac{E \vdash (\forall X)\, t = t'}{E \vdash (\forall X)\, t' = t}$$

- Transitividade.

$$\frac{E \vdash (\forall X)\, t = t' \quad E \vdash (\forall X)\, t' = t''}{E \vdash (\forall X)\, t = t''}$$

[2] A seguinte descrição segue a descrição de MEL em [31] e [13].

- Congruência.

$$\frac{E \vdash (\forall X)\, t_1 = t_1' \quad \cdots \quad E \vdash (\forall X)\, t_n = t_n'}{E \vdash (\forall X)\, f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}$$

- Pertinência.

$$\frac{E \vdash (\forall X)\, t = t' \quad E \vdash (\forall X)\, t : s}{E \vdash (\forall X)\, t' : s}$$

- *Modus ponens* para equações.[3] Dada uma sentença:

$$(\forall X)\, t = t' \Leftarrow u_1 = v_1 \wedge \cdots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \cdots \wedge w_m : s_m$$

no conjunto E de axiomas, e dado uma atribuição K-tipada $\theta : X \to T_\Sigma(Y)$ então, para $1 \leq i \leq n$ e $1 \leq j \leq m$, em que a partir de θ podemos obter sua extensão única a um Σ-homomorfismo $\overline{\theta} : T_\Sigma(X) \to T_\Sigma(Y)$ (ver [13, seção 2] para maiores detalhes).

$$\frac{E \vdash (\forall Y)\, \overline{\theta}(u_i) = \overline{\theta}(v_i) \quad E \vdash (\forall Y)\, \overline{\theta}(w_j) : s_j}{E \vdash (\forall X)\, \overline{\theta}(t) = \overline{\theta}(t')}$$

- *Modus ponens* para axiomas de pertinência. Dada uma sentença:

$$(\forall X)\, t : s \Leftarrow u_1 = v_1 \wedge \cdots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \cdots \wedge w_m : s_m$$

no conjunto E de axiomas, e dado uma atribuição K-tipada $\theta : X \to T_\Sigma(Y)$ then, for $1 \leq i \leq n$ and $1 \leq j \leq m$.

$$\frac{E \vdash (\forall Y)\, \overline{\theta}(u_i) = \overline{\theta}(v_i) \quad E \vdash (\forall Y)\, \overline{\theta}(w_j) : s_j}{E \vdash (\forall X)\, \overline{\theta}(t) : s}$$

Conforme mencionado anteriormente, exemplos práticos do uso de MEL são exibidos na seção 2.3.

Uma *teoria de reescrita*[4] é uma tupla $\mathcal{R} = (\Omega, E, R)$ onde (Ω, E) é uma teoria MEL,

[3] As regras para *modus ponens* para equações e axiomas de pertinência são usualmente exibidas juntas; optamos pela separação em duas regras distintas para evitar uma única regra desnecessariamente complexa.

[4] Usamos aqui a definição original de teorias de reescritas e não a versão *generalizada* definida por Bruni e Meseguer em [32] já que *Maude MSOS Tool* não faz uso da construção de operadores *congelados* (*frozen*).

como descrito acima; R é um conjunto de regras rotuladas, condicionais de reescrita universalmente quantificadas da forma ([32, seção 1.1])

$$l : t \to t' \Leftarrow (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \to w_k')$$

com as seguintes regras de dedução:

- Reflexividade.

$$\frac{}{(\forall X)\, t \to t}$$

- Transitividade.

$$\frac{(\forall X)\, t_1 \to t_2 \quad (\forall X)\, t_2 \to t_3}{(\forall X)\, t_1 \to t_3}$$

- Igualdade.

$$\frac{(\forall X)\, u \to v \quad E \vdash (\forall X)\, u = u' \quad E \vdash (\forall X)\, v = v'}{(\forall X)\, u' = v'}$$

- Congruência. Para cada $f : k_1 \cdots k_n \to k$ em Σ

$$\frac{(\forall X)\, t_{j_1} \to t'_{j_1} \quad \cdots \quad (\forall X)\, t_{j_m} \to t'_{j_m}}{(\forall X)\, f(t_1, \ldots, t_{j_1}, \ldots, t_{j_m}, \ldots, t_n) \to f(t_1, \ldots, t'_{j_1}, \ldots, t'_{j_m}, \ldots, t_n)}$$

- Substituição aninhada[5] Considere as substituições finitas $\theta, \theta' : X \to T_\Sigma(Y)$. Dada uma regra de reescrita com $1 \leq i \leq n$.

$$(\forall X)\, l : t \to t' \Leftarrow \bigwedge_i t_i \to t'_i$$

Para $1 \leq i \leq n$ e $x \in X$:

$$\frac{(\forall Y)\, \theta(t_i) \to \theta(t'_i) \quad (\forall Y)\, \theta(x) \to \theta'(x)}{(\forall Y)\, \theta(t) \to \theta'(t)'}$$

[5]Aqui seguimos a descrição da regra como definida em [32], considerando apenas reescritas nas condições; de fato, uma versão mais geral da regra leva em consideração também equações e axiomas de pertinência nas condições.

Esta regra diz que, dada uma regra $r \in R$ e duas atribuições θ, θ' para suas variáveis de forma que para cada $x \in X$ temos $\theta(x) \to \theta'(x)$, então r pode ser concorrentemente aplicada para as reescritas de seus argumentos, uma vez que as condições de r sejam satisfeitas no estado inicial definido por θ.

Lógica de reescrita provê uma leitura computacional das suas regras de inferência que permite a especificação de sistemas concorrentes:[6] *reflexividade* representa o fato de um sistema ter *transições inócuas*; *igualdade* representa o fato de que os *estados* de um sistema concorrente são iguais *modulo* o conjunto de equações E; *congruência* é uma forma geral de *paralelismo lateral* onde os argumentos do operador f podem evoluir em paralelo; *substituição aninhada* combina uma transição atômica no topo usando uma regra com concorrência aninhada na substituição; *transitividade* é composição seqüencial.

É importante discutir como uma teoria de reescrita poderia ser *executável* eficientemente por uma implementação. Para que isto ocorra alguns requisitos precisam ser satisfeitos ([34]): o conjunto de equações E deve permitir a sua decomposição numa união $E = E_0 \cup A$, com A sendo um conjunto de axiomas equacionais como associatividade, comutatividade e identidade para os quais um algoritmo eficiente de casamento de padrões *modulo* A exista. Além disto, E_0 deve ser confluente e terminante (ou seja, aplicando as equações em E_0 *modulo* A a um termo t, iremos, após um número finito de reescritas chegar a uma única forma normal). Já as regras em R devem ser *coerentes* [35] com E_0 *modulo* A, o que significa que, de forma a reescrever em classes de equivalência *modulo* E, podemos sempre simplifcar um termo usando as equações para sua forma canônica, e então reescrevê-lo com uma regra em R. Finalmente, as regras em R devem ser *admissíveis* ([36]), o que significa, intuitivamente, que não deve haver metavariáveis livres.

Finalmente, lógica de reescrita é reflectiva no sentido em que sua metateoria pode ser representada no nível objeto de uma maneira consistente, de modo que este nível objeto simule corretamente os aspectos metateoréticos relevantes [37]. Em outras palavras, existe uma teoria de reescrita finita $\mathcal{U}$ que pode simular qualquer outra teoria finitamente representável $\mathcal{R}$ da seguinte forma: dados dois termos t, t' em $\mathcal{R}$, existem termos correspondentes $(\overline{\mathcal{R}}, \overline{t})$ e $(\overline{\mathcal{R}}, \overline{t'})$ em $\mathcal{U}$ de forma que temos:

$$\mathcal{R} \vdash t \to t' \quad \Leftrightarrow \quad \mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}}, \overline{t'})$$

Dado que $\mathcal{U}$ é ela mesma representável, chegamos à chamada "torre de reflexão."

[6] Baseado em [33].

$$\mathcal{R} \vdash \mathtt{t} \to \mathtt{t}' \quad \Leftrightarrow \quad \mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{\mathtt{t}}) \to (\overline{\mathcal{R}}, \overline{\mathtt{t}'}) \quad \Leftrightarrow \quad \mathcal{U} \vdash (\overline{\mathcal{U}}, \overline{(\overline{\mathcal{R}}, \overline{\mathtt{t}})}) \to (\overline{\mathcal{U}}, \overline{(\overline{\mathcal{R}}, \overline{\mathtt{t}'})}) \cdots$$

Esta característica de lógica de reescrita e sua relação com o interpretador Maude é discutida na seção 2.3.2.

2.3 Maude

O nome *Maude* refere-se tanto à linguagem e sua implementação [26] de alta performance escrita em C++ em Lógica de Reescrita capaz de processamentos da ordem de milhões de reescritas por segundo (veja o apêndice F para um exemplo). Até a versão 2.1.1, Maude é um interpretador, mas um compilador está em produção, o que promete elevar este número para dezenas de milhões de reescritas. De forma a simplificar esta seção, optamos por descrever os aspectos da linguagem e do interpretador relevantes à implementação e ao uso de *Maude MSOS Tool*. A descrição completa pode ser encontrada em [26].

Maude implementa teorias de lógica de reescrita e lógica equacional de pertinência através dos chamados *módulos de sistema* (*system modules*) e *módulos funcionais* (*functional modules*), respectivamente. Módulos de sistema são criados usando-se a sintaxe '`mod` n `is` D `endm`' e módulos funcionais com a sintaxe '`fmod` n `is` D `endfm`', onde n é o nome do módulo e D as suas declarações.

A importação de módulos em Maude é feita usando uma das seguintes construções: '`including`' (ou, abreviadamente, '`inc`'), '`extending`' ('`ex`') e '`protecting`' ('`pr`'). A diferença entre os três tipos de inclusão é se *junk* ou *confusion* são permitidos na importação.[7] Informalmente, em especificações algébricas, *no confusion* ("sem confusão") é o requisito que termos diferentes denotam coisas diferentes e *no junk* ("sem lixo") significa que a álgebra é *mínima*, ou seja tem apenas os elementos necessários. Ao incluir um módulo através do modo '`protecting`', não são permitidos *junk* ou *confusion*. Por exemplo: se importarmos '`BOOL`' em um módulo '`FOO`' no via '`protecting`', estamos assumindo que nenhuma nova constante do *sort* '`Bool`' será criada (*no junk*), nem nenhum novo significado será adicionado ao módulo '`BOOL`', tal como fazendo com que as constantes '`true`' e '`false`' sejam iguais. A forma mais fraca de inclusão, '`extending`' permite *junk*, mas não *confusion*. (É útil em nosso caso quando a assinatura de um módulo está sendo estendida por um outro módulo com novas constantes, como é o caso de uma declaração

[7] Traduzindo livremente, "lixo" ou "confusão".

modular da sintaxe de uma linguagem de programação.) A forma mais geral, onde *junk* e *confusion* são permitidos, é através da construção '`including`'. Maude não verifica se uma determinada importação respeita os requisitos de sua construção, dado que isto necessitaria a capacidade de prova de teoremas.

Criamos a assinatura Σ em Maude declarando os sorts com a construção '`sort`', as relações de subsorts com '`subsort`', operadores com '`op`' e axiomas de pertinência com '`mb`' e '`cmb`'.

Vamos exemplificar estes conceitos modelando a formação de *palavras* e *letras* numa linguagem. Começamos com os conceitos simples de letras, vogais e consoantes, representados, respectivamente, pelos sorts '`Letra`', '`Vowel`' e '`Consoante`'.

```
sort Letra .
sort Vogal .
sort Consoante .
```

Para dar a ordem dos sorts no conjunto parcialmente ordenado $(S_k, \leq)$, usamos a construção '`subsort`'. Neste exemplo, vogais e consoantes são letras.

```
subsort Vogal < Letra .
subsort Consoante < Letra .
```

Os operadores que são parte da assinatura $\{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ são definidos com '`op`' de acordo com a seguinte sintaxe:

```
op o : w̄ -> k [A] .
```

onde `o` é o nome do operador, $\bar{w} = w_1 \cdots w_n$ são os *sorts* (ou *kinds*) do domínio, `k` é o *sort* (ou *kind*) da imagem, `A` os atributos equacionais: '`assoc`' define operadores associativos, '`comm`' define operadores comutativos e '`id:t`' define o termo `t` como a identidade do operador sendo definido. Formalmente, como discutimos na seção 2.2, isto significa que as reescritas e reduções acontecerão *modulo* estes atributos. A construção '`ops`' é uma variante de '`op`' em que vários operadores podem ser definidos de uma só vez se eles tiverem os mesmos *sorts* de domínio e imagem. No exemplo que segue, não há *sorts* de domínio—os operadores são *constantes*. A construção '`constructor`' (ou '`ctor`', abreviadamente) não é um atributo equacional, mas um indicador que este operador é um *construtor* de termos. Construtores definem a *estrutura* dos termos na especificação, ao passo que os demais operadores computam novos termos a partir de seus argumentos.

```
ops a e i o u : -> Vogal [constructor] .
ops b c d f g h j k l m n
      p q r s t v w x y z : -> Consoante [constructor] .
```

Se usarmos o comando interno de Maude '`reduce`' (em essência aplica as equações em E para um Σ-termo e é descrito em detalhe na seção 2.3.1) podemos verificar que a vogal '`a`' é também uma letra, como esperado. O operador '`t : : s`' é definido internamente como um predicado que é '`true`' se `t` tem *sort* `s`.

```
Maude> reduce a :: Letra .
reduce in WORDS : a :: Letra .
result Bool: true
```

A constante '`a`' não é uma consoante, como a seguinte execução do comando '`reduce`' mostra.

```
Maude> reduce a :: Consoante .
reduce in WORDS : a :: Consoante .
result Bool: false
```

Como os *sorts* '`Letra`', '`Vogal`' e '`Consoante`' são todos relacionados entre si, eles formam um *componente conexo* e pertencem todos ao mesmo *kind*. Este *kind* não é nomeado explicitamente: seu nome é obtido a partir do nome de um dos *sorts* que pertencem ao componente conexo entre colchetes, tal como '`[Vogal]`'. O nome padrão escolhido por Maude é o *sort* maior na hierarquia de *sorts*, no exemplo, '`[Letra]`'.

Vamos agora adicionar o conceito de *palavra* à nossa simples especificação. Começamos definindo um novo *sort* '`Palavra`'. Para simplificar esta exposição, nesta especificação uma única letra é uma palavra "trivial."

```
sort Palavra .
subsort Letra < Palavra .
```

Usamos agora o operador de justaposição para criar um operador sem nome, onde um novo termo é criado colocando-se os seus argumentos lado a lado.

```
op __ : [Palavra] [Palavra] -> [Palavra] [assoc] .
```

O uso do atributo equacional '`assoc`' significa que este operador é associativo, como dissemos anteriormente. Intuitivamente, isto significa que podemos escrever '`a b c`' ao invés de '`(a b) c`' e assim por diante. Ao escrever um termo tal como '`c b v n`', este será identificado como '`[Palavra]`'. Repare que o operador de justaposição foi definido sobre o *kind* '`[Palavra]`' com um propósito: lembre-se que em MEL um termo com um *kind* mas sem um *sort* é considerado um *termo incorreto*. Isto se encaixa bem com nosso objetivo, já que queremos que *algumas* seqüências de letras sejam palavras, mas não todas.

Vamos agora adicionar a capacidade de identificar palavras de sequências sem sentido à nossa especificação. Fazemos aqui uso da construção '`mb`' de Maude para declarar um axioma de pertinência, com a seguinte sintaxe:

```
mb t : s .
```

onde `t` é um termo e `s` é um sort. Usamos esta construção para identificar as seqüências de letras que são palavras na língua.

```
mb  a a b o r a       : Palavra .
mb  a a c h e n s e   : Palavra .
mb  a a l e n i a n o : Palavra .
mb  a a r i a n o     : Palavra .
mb  a a r u           : Palavra .
          ...
```

Se reduzirmos '`a x q`', veremos que Maude nos diz que é do *kind* '`[Palavra]`' (sem *sort*), enquanto que a redução de '`c a r r o`' dá o *sort* correto '`Palavra`', que passou a ser o maior *sort*.

```
reduce in WORDS : a x q .
result [Palavra]: a x q

reduce in WORDS : c a r r o .
result Palavra: c a r r o
```

Com isto, exemplificamos os conceitos de *sorts*, *kinds* e operadores com atributos equacionais; vamos agora exemplificar o uso de equações e regras de reescrita. Iremos assumir a existência de um módulo interno '`INT`' que define os inteiros pelo *sort* '`Int`'.

Começamos definindo um *conjunto de inteiros* (‘`IntSet`’), que é representado por um operador associativo-comutativo. Este conjunto de inteiros tem ‘`null`’ como identidade.

```
sort IntSet .
subsort Int < IntSet .

op null : -> IntSet .
op __ : IntSet IntSet -> IntSet [assoc comm id: null] .
```

Como está definido, o conjunto de inteiros ‘`IntSet`’ não está correto, pois ele permite a repetição de elementos; adicionaremos, portanto, uma equação que elimina elementos duplicados de um ‘`IntSet`’.

Metavariáveis em especificações Maude devem ser explicitamente declaradas antes de serem usadas, de acordo com a seguinte sintaxe:

```
var ν : s .
```

onde ν é o nome da metavariável e `s` seu sort. Uma forma alternativa que evita esta declaração prévia é usá-las explicitamente nos axiomas e regras usando a sintaxe ‘ν`:s`’.

Equações incondicionais em Maude são escritas com a seguinte sintaxe:

```
eq [L] : t = t' .
```

onde `t` e `t'` são termos que pertencem ao mesmo *kind*, e `L` é um rótulo opcional.

No exemplo abaixo, inicialmente criamos uma metavariável ‘`I`’ sobre ‘`Int`’; em seguida criamos uma equação que especifica que dois inteiros repetidos devem ser removidos e uma única cópia deve ser mantida. A equação abaixo poderia ter sido descrita com a sintaxe alternativa: ‘`eq I:Int I:Int = I:Int .`’, evitando assim a declaração prévia da metavaríavel.

```
var I : Int .
eq I I = I .
```

Apenas esta equação é suficiente para removermos todos os elementos repetidos de um conjunto de inteiros dado que Maude reescreve *modulo* classes de equivalência—neste

caso, a classe de equivalência gerada pelo operador associativo-comutativo '`__`'. A regra de dedução de *Congruência* de MEL faz com que a equação seja aplicável quantas vezes forem necessárias dentro de um termo. Por exemplo, um termo como '`1 2 1 2`' será casado duas vezes com o padrão '`I I`': o primeiro casamento será '`1 1`' e o segundo '`2 2`'.

```
reduce in INTEGER : 1 2 1 6 2 1 2 1 2 1 2 .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result IntSet: 1 2 6
```

O atributo de identidade do operador '`__`' pode ser exemplificado com o seguinte exemplo de redução, onde a identidade '`null`' é, como se espera, removida do conjunto de inteiros.

```
reduce in INTEGER : 1 2 3 null 4 3 1 null 1 3 .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result IntSet: 1 2 3 4
```

Para demonstrar o uso de regras de reescrita iremos criar uma operação que seleciona, de maneira não-determinística, um inteiro de um conjunto de inteiros. Começamos pelo operador.

```
op select : IntSet -> Int .
```

Adicionamos agora a regra. Em Maude, regras não condicionais devem ser escritas de acordo com a seguinte sintaxe:

```
rl [L] : t => t' .
```

onde `t` e `t'` são termos que pertencem ao mesmo *kind*, e `L` é um rótulo opcional.

A regra '`select`' selecionará não-deterministicamente um inteiro de um conjunto de inteiros devido ao casamento associativo-comutativo do padrão '`I S`'.

```
rl [select] : select(I S) => I .
```

O comando '`rewrite`' (seção 2.3.1) rescreve um dado termo `t` é que nenhuma regra de aplique. No exempo abaixo, ele aplicará apenas uma reescrita ao termo '`select(...)`',

dado que ele será reescrito para um inteiro, ao qual nenhuma regra se aplica. (A razão pela qual o *sort* de '`1`' é '`NzNat`' se dá porque Maude sempre exibe o nome do menor *sort* aplicável a um termo. O módulo '`INT`' de fato importa outros módulos que formam uma hierarquia de *sorts* envolvendo '`NzNat`', os naturais maiores que zero, '`Nat`', os naturais, '`NzInt`', os inteiros acima de zero e '`Int`'. O menor *sort* aplicável a '`1`' é '`NzNat`' neste caso.)

```
rewrite in INTEGER : select(1 2 1 6 2 1 2 1 2 1 2) .
rewrites: 9 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 1
```

A chance da regra selecionar '`1`', '`2`' ou '`6`' é a *mesma*. Isto porque as equações são aplicadas *antes* das regras, devido ao requisito de *coerência*. Assim, o termo '`1 2 1 6 2 1 2 1 2 1 2`' será reduzido para '`1 2 6`' e a regra será aplicada sobre este termo resultante. Na seção 2.3.1 veremos como as ferramentas presentes em Maude podem ser usadas para *buscar* todos os resultados possíveis de especificações não-determinísticas.

Vamos concluir esta seção com um comentário sobre condições. Optamos até o momento por fazer os exemplos simples para simplificar a exposição; contudo Maude também suporta o uso de axiomas de pertinência, equações e regras *condicionais*, como definimos formalmente na seção 2.2. Eles são definidos de acordo com a seguinte sintaxe.

Para axiomas de pertinência condicionais.

```
cmb t : s if C .
```

Para equações condicionais.

```
ceq t = t' if C .
```

Para regras de reescrita condicionais.

```
crl t => t' if C .
```

A condição C é ou uma conjunção de condições combinadas com o operador '`_/\_`', ou uma das seguintes opções:

- Equações comuns '`t = t'`', que são satisfeitas se e somente se as formas canônicas de `t` e `t'` são iguais *modulo* os atributos equacionais especificados nos operadores que compõem `t` e `t'`;
- Equações booleanas abreviadas, tais como `p(t)`, que são uma abreviação da equação '`p(t) = true`'. Este um grande número de predicados predefinidos disponíveis, tais como: igualdade (`_==_`), desiguladade (`_=/=_`), pertinência (`_::_`, etc.
- "Equações de casamento" (*matching equations* [26]), escritas como '`t := t'`', que são equações comuns, com alguns requisitos operacionais. Em essência, estas equações são usadas para instanciar novas metavariáveis através de casamento dos padrões do termo à esquerda com o termo à direita;
- Reescritas '`t => t'`', que são satisfeitas se existir uma prova de zero ou mais reescritas do termo `t` para o termo `t'`.

Reescritas condicionais devem ser apenas usadas em regras condicionais; de resto todos os tipos de condições podem ser usadas em axiomas de pertinência, equações e regras.

2.3.1 Ferramentas

Nós já vimos alguns exemplos das ferramentas disponíveis no interpretador Maude, especificamente a capacidade de reduzir e reescrever termos. Esta seção descreve estes comandos em mais detalhes, além de descrever os comandos para *buscas em largura* e *verificações de modelos*.

2.3.1.1 Reduzindo e reescrevendo termos

Começamos pelo comando '`reduce`', abreviado '`red`'. Ele recebe um argumento, um termo `t`, e tenta reduzi-lo para uma *formal normal*, usando todas as equações aplicáveis até que nenhuma possa ser aplicada.

A sintaxe é '`reduce t`', onde `t` é um termo a ser reduzido. Podemos opcionalmente especificar que esta redução se dará num módulo específico `M` escrevendo: '`reduce in M : t`'. Por exemplo:

```
Maude> red in NAT : 100 + 50 .
```

```
reduce in NAT : 50 + 100 .
rewrites: 1 in 0ms cpu (0ms real)
result NzNat: 150
```

O comando '`rewrite t`', abreviado '`rew`' é similar, mas tenta reescrever um termo `t` inicialmente reduzindo-o à sua forma canônica, de acordo com as equações em `E`, e então repetidas vezes aplicando as regras de reescritas em `R` até que nenhuma seja aplicável. Diferente de equações que devem ser Church-Rosser e terminantes, regras de reescritas não têm estes requisitos e podem levar a não-determinismo e não-terminação. Para lidar com esta possibilidade, o comando também permite, opcionalmente, estabelecer um limite superior `n` no número de reescritas a serem feitas, '`rewrite [n] t`'.

Considere o seguinte módulo de sistema, por exemplo, que incrementa o argumento, de *sort* '`Nat`', do operador '`counter`', de *sort* '`Counter`'. Este módulo também mostra um exemplo de uma regra de reescrita condicional: a regra '`inc`' será aplicável a '`counter(n)`' se `n` for menor que 1000.

```
mod COUNTER is protecting NAT .
 sort Counter .
 op counter : Nat -> Counter .

 var n : Nat .

 crl [inc] : counter (n) => counter (n + 1)
 if n < 1000 .
endm
```

Executando um comando '`rewrite`' sem limite de reescritas, atingimos o máximo valor possível do contador, representado pelo termo '`counter(1000)`'.

```
Maude> rew counter(0) .
rewrite in COUNTER : counter(0) .
rewrites: 3001 in 10ms cpu (10ms real) (300100 rewrites/second)
result Counter: counter(1000)
```

Contudo, se limitarmos a reescrita a um número `n`, coincidentemente, atingimos o valor `n` como argumento do contador, já que, neste módulo, uma reescrita representa o incremento de um ao valor do contador.

```
Maude> rew [100] counter(0) .
rewrite [100] in COUNTER : counter(0) .
rewrites: 300 in 0ms cpu (0ms real) (~ rewrites/second)
result Counter: counter(100)
```

2.3.1.2 Buscando por estados

Outra ferramenta disponível é a busca em largura que é feita pelo comando '`search`'. Essencialmente, este comando procura por uma *prova de reescrita* de um termo a um padrão final aplicando as regras de dedução do cálculo de reescrita. A sintaxe do comando é a seguinte (o que está entre { e } é opcional):

```
search {[b]} {in m :} t R p {such that C}
```

onde `b` é um limite superior no número de soluções retornadas pelo comando, por *default* o comando retorna todas as possíveis soluções; `m` é o módulo no qual a busca será feita, cujo *default* é o módulo corrente; `t` é o estado inicial de onde a busca começará; `p` é o padrão do estado final; `R` é a *relação* entre `t` e `p`, e pode ser uma das seguintes opções:

- '`=>1`': prova com um passo de reescrita;
- '`=>+`': prova com um ou mais passos de reescrita;
- '`=>*`': prova com zero ou mais passos de reescrita;
- '`=>!`': apenas estados finals são permitidos.

Uma condição opcional `C` pode ser especificada a ser satisfeita pela prova de reescrita.

Para exemplificar isto, vamos retornar ao nosso primeiro exemplo onde um número é selecionado não-deterministicamente de um conjunto de inteiros.

```
mod INTEGER is protecting INT .
 sort IntSet .
 subsort Int < IntSet .

 op null : -> IntSet .
 op __ : IntSet IntSet -> IntSet [assoc comm id: null] .
```

```
 var S : IntSet .
 var I : Int .

 eq I I = I .

 op select : IntSet -> Int .

 rl select(I S) => I .
endm
```

Se quisermos buscar por todos os termos possíveis que são atingíveis começando com '`select(1 2 1 6 2 1 2 1 2 1 2)`' devemos usar o padrão '`S:IntSet`'. Se quisermos uma prova que inclua zero ou mais passos de reescrita, devemos usar a relação '`=>*`'. Verifique na saída abaixo que o padrão '`S:IntSet`' é casado com todos os estados possíveis atingíveis em zero ou mais passos de reescritas a partir do termo '`select(1 2 1 6 2 1 2 1 2 1 2)`'. Lembramos que as regras de reescritas em Maude são *coerentes*, e todos os termos são reduzidos a uma forma normal antes das regras de reescritas serem aplicadas, ou seja, o termo '`select(1 2 1 6 2 1 2 1 2 1 2)`' primeiramente é reduzido a '`select(1 2 6)`' antes que a busca começe.

```
search in INTEGER : select(1 2 1 6 2 1 2 1 2 1 2) =>* S .

Solution 1 (state 0)
states: 1  rewrites: 8 in 0ms cpu (0ms real)
S --> select(1 2 6)

Solution 2 (state 1)
states: 2  rewrites: 9 in 0ms cpu (0ms real)
S --> 1

Solution 3 (state 2)
states: 3  rewrites: 10 in 0ms cpu (0ms real)
S --> 2

Solution 4 (state 3)
```

```
states: 4  rewrites: 11 in 0ms cpu (0ms real)
S --> 6

No more solutions.
states: 4  rewrites: 11 in 0ms cpu (10ms real)
```

Se quisermos apenas estados finais, ou seja, estados onde nenhuma regra de reescrita é aplicável, devemos usar a relação '`=>!`'.

```
search in INTEGER : select(1 2 1 6 2 1 2 1 2 1 2) =>! S .

Solution 1 (state 1)
states: 4  rewrites: 11 in 0ms cpu (0ms real)
S --> 1

Solution 2 (state 2)
states: 4  rewrites: 11 in 0ms cpu (0ms real)
S --> 2

Solution 3 (state 3)
states: 4  rewrites: 11 in 0ms cpu (0ms real)
S --> 6

No more solutions.
states: 4  rewrites: 11 in 0ms cpu (0ms real)
```

2.3.1.3 Verificação de modelos de especificações

Maude contém também um verificador de modelos que suporta fórmulas da *lógica temporal linear* (*linear temporal logic*, LTL). Esta seção exibe uma rápida descrição das suas capacidades de verificação e como fórmulas LTL são codificadas em Maude. A informação presente nesta seção é baseada em [26, Chapter 9].

Vamos descrever de maneira indutiva o conjunto de fórmulas da lógica proposicional temporal linear LTL(AP) sobre um conjunto AP de proposições atômicas. Iremos exibir também a sua assinatura concreta definida pelo módulo '`MODEL-CHECKER`' em Maude.

- $\top \in$ LTL(AP) é sempre satisfeita, escrita como '`True`';

- se $\varphi \in \mathrm{LTL}(\mathsf{AP})$, então $\bigcirc\varphi \in \mathrm{LTL}(\mathsf{AP})$ é o operador *next*, que é satisfeito se φ é satisfatível no próximo passo da computação, escrito como '`O` φ';
- se $\varphi, \psi \in \mathrm{LTL}(\mathsf{AP})$, então $\varphi\,\mathcal{U}\,\psi \in \mathrm{LTL}(\mathsf{AP})$ é i operador *strong until*, que é satisfeito, se durante a computação, φ é válida, até que ψ se torne válida, escrito como 'φ `U` ψ';
- *proposições atômicas*, se $\mathtt{p} \in \mathsf{AP}$ então $\mathtt{p} \in \mathrm{LTL}(\mathsf{AP})$, e são definidas por operadors em Maude cujo *sort* de imagem é '`Prop`';
- *conectivos booleanos*, se $\varphi, \psi \in \mathrm{LTL}(\mathsf{AP})$ então $\neg\varphi$ e $\varphi \vee \psi$ pertencem a $\mathrm{LTL}(\mathsf{AP})$, e são escritos como '`~_`' e '`_or_`'.

A tabela 2.1 exibe um conjunto de fórmulas derivadas deste conjunto primitivo.

Nome	*Fórmula*	*Fórmula equivalente*	*Sintaxe*
falso	$\bot$	$\neg\top$	`False`
conjução	$\varphi \wedge \psi$	$\neg((\neg\varphi) \vee (\neg\psi))$	φ `/\` ψ
implicação	$\varphi \rightarrow \psi$	$(\neg\varphi) \vee \psi$	φ `|->` ψ
eventually	$\Diamond\varphi$	$\top\,\mathcal{U}\,\varphi$	`<>`φ
henceforth	$\Box\varphi$	$\neg\Diamond\neg\varphi$	`[]`φ
release	$\varphi\,\mathcal{R}\,\psi$	$\neg((\neg\varphi)\,\mathcal{U}\,(\neg\psi))$	φ `R` ψ
unless	$\varphi\,\mathcal{W}\,\psi$	$(\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)$	φ `W` ψ
leads-to	$\varphi \rightsquigarrow \psi$	$\Box(\varphi \rightarrow (\Diamond\psi))$	φ `|->` ψ
strong implication	$\varphi \Rightarrow \psi$	$\Box(\varphi \rightarrow \psi)$	φ `=>` ψ
strong equivalence	$\varphi \Leftrightarrow \psi$	$\Box(\varphi \leftrightarrow \psi)$	φ `<=>` ψ

Tabela 2.1: Fórmulas LTL derivadas das fórmulas primitivas

Estruturas de Kripke são os modelos naturais para lógica proposicional linear temporal. Uma estrutura de Kripke é essencialmente um sistema de transição total e desprovido de rótulos para o qual um conjunto de predicados unários foi adicionado aos estados. Formalmente, é uma tripla $\mathcal{A} = (\mathsf{A}, \rightarrow_{\mathcal{A}}, \mathsf{L})$, onde A é um conjunto de estados, $\rightarrow_{\mathcal{A}}$ é uma relação binária total em A, e $\mathsf{L} : \mathsf{A} \rightarrow \mathcal{P}(\mathsf{AP})$ é uma função, chamada de *função de rotulamento* (*labelling function*), associando a cada estado $\mathtt{a} \in \mathsf{A}$ o conjunto $\mathsf{L}(\mathtt{a})$ das proposições atômicas em AP que são satisfeitas no estado $\mathtt{a}$. Num módulo de sistema que especifica uma teoria de reescrita $\mathcal{R} = (\Omega, \mathsf{E}, \mathsf{R})$, a estrutura de Kripke contém como conjunto de estados A o conjunto $\mathsf{T}_{\Omega/\mathsf{E},\mathsf{k}}$ que é o conjunto de *termos canônicos* do *kind* k, a relação de transição $\rightarrow_{\mathcal{A}}$ é a transição de reescrita *de um passo* de termos do *kind* k e a função de rotulamento $\mathsf{L}(\mathtt{a})$ forma o conjunto de proposições atômicas definidas equacionalmente sobre termos do *kind* k que são satisfeitas no estado $\mathtt{a} \in \mathsf{A}$.

Procedemos agora com um exemplo simples, uma máquina de estados com elementos ‘a’, ‘b’, ‘c’, ‘d’ e ‘f’, do *sort* ‘Elt’. Estados são criados com o operador ‘st_’, que recebe como único argumento um ‘Elt’. As transições desta máquina de estados são dadas pelas regras de reescritas rotuladas no módulo ‘STATE-MACHINE’ abaixo. As duas equações estão presentes somente para demonstrar que, de fato, o estado de espaço é formado apenas por formas canônicas, ou seja, ‘a’–‘f’. A seção 6.3 e o apêndice E contêm diversos exemplos do verificador de modelos de Maude para uma grande variedade de especificações

```
mod STATE-MACHINE is
 sorts St Elt .
 op st_ : Elt -> St .
 ops a b c d e f BB CC : -> Elt .
 eq BB = b .
 eq CC = c .
 rl [a->b] : st a => st BB .
 rl [a->c] : st a => st CC .
 rl [b->a] : st b => st a .   rl [c->d] : st c => st d .
 rl [c->e] : st c => st e .   rl [d->f] : st d => st f .
 rl [f->a] : st f => st a .
endm
```

Para usarmos o verificador de modelos, devemos primeiro estabelecer nosso *espaço de estados* da especificação. Neste caso, é claramente o espaço definido pelo *sort* ‘St’ com o operador ‘st_’. No módulo ‘CHECK-STATE-MACHINE’ abaixo, após incluir o módulo ‘MODEL-CHECKER’, criamos a relação de *subsort* entre ‘St’ e ‘State’, onde este último é um *sort* definido no módulo ‘MODEL-CHECKER’ que representa o espaço de estados que será explorado pelo algoritmo de verificação de modelos.

```
mod CHECK-STATE-MACHINE is
 protecting STATE-MACHINE .
 including MODEL-CHECKER .

 subsort St < State .
 ...
endm
```

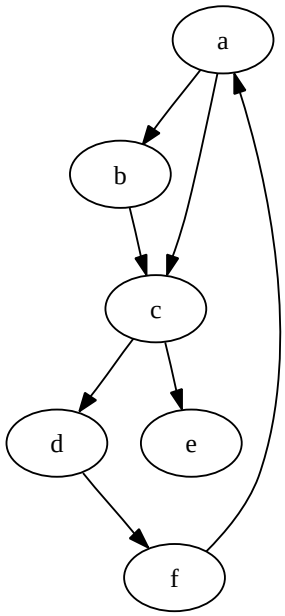

Figura 2.2: Estrutura Kripke (simplificada) para '`STATE-MACHINE`'

Estamos agora prontos para definir uma proposição simples a ser verificada. Esta deve ser um operador cujo *sort* de imagem é '`Prop`'. Devemos definir equacionalmente a condição em que esta proposição é satisfeita. Isto é feito através de uma equação envolvendo o operador:

$$\texttt{\_:=\_} : \texttt{State} \times \texttt{Formula} \rightharpoonup \texttt{Bool}$$

onde '`Formula`' ou é uma proposição, uma das fórmulas presentes em LTL($\mathcal{AP}$), descritas no início desta seção, ou uma das fórmulas da tabela 2.1.

No exemplo abaixo, criamos uma proposição '`in s`' que é satisfeita sempre quando o estado atual é `s`. Dado que operador '`_:=_`' é parcial, não há necessidade de especificar quando um estado é falso.

```
var n : Elt .
op in_ : Elt -> Prop .
eq st n |= in n = true .
```

A figura 2.2 mostra a estrutura Kripke associada ao módulo de sistema '`STATE-MACHINE`'. Os estados são termos canônicos '`a`'–'`f`', as setas representam as relações de um passo de reescrita entre cada estado. Apesar de não termos explicitado na figura, cada estado `s` contém uma fórmula proposicional associada '`in s`'. Por exemplo, no es-

tado `a`, a proposição '`in a`' é verdadeira, enquanto que '`in b`', '`in c`', '`in d`' e '`in f`' são falsas.

Para verificar uma fórmula LTL precisamos do seguinte operador:

$$\texttt{modelCheck} : \texttt{State} \times \texttt{Formula} \rightharpoonup \texttt{ModelCheckResult}$$

onde o primeiro parâmetro é o estado inicial e o segundo é a fórmula a ser verificada. A imagem '`ModelCheckResult`' é ou '`true`' ou um contra-exemplo à fórmula dada.

Como um exemplo de verificação de uma fórmula LTL, vamos verificar que se o estado '`b`' *nunca ocorrer* ('`~ <> in b`'), então eventualmente o estado '`f`' ocorrerá ('`<> in f`').

```
reduce in CHECK-STATE-MACHINE :
 modelCheck(st a, ~ <> in b -> <> in f) .
rewrites: 21 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult:
 counterexample({st a,'a->c} {st c,'c->e}, {st e, deadlock})
```

A verificação falha, dando um contra-exemplo, um par de listas de transições formado pelo operador '`counterexample(t,t')`', onde `t` corresponde a um caminho finito começando no estado inicial ('`st a`') e `t'` descreve um *ciclo* (estado '`st e`' é um beco sem saída e pode apenas reescrever para si mesmo através da regra de *Reflexividade* do cálculo de reescrita, veja a seção 2.2). Os contra-exemplos têm esta forma dado que se uma fórmula LTL φ não for satisfeita por uma estrutura de Kripke, é sempre possível encontrar um contra-exemplo para φ que tem a forma de um caminho de transições seguido por um ciclo (veja [26, capítulo 9]).

2.3.2 Programação no metanível

Os aspectos reflexivos e universais da lógica de reescritas discutidos ao final da seção 2.2 são efetivamente implementados pelo módulo '`META-LEVEL`' em Maude. Este módulo define uma metarrepresentação de todas as construções de Maude, como *sorts*, constantes, (meta)variáveis, termos, módulos, equações, regras de reescrita, etc. Descrevemos nesta seção as funções que são relevantes para o entendimento da implementação de *Maude MSOS Tool*.

As funções "de ascensão" (*up functions*) convertem termos no modo objeto para o

metanível, ou seja, criam metarrepresentações de termos-objeto. A função '`upModule(q)`' produz a metarrepresentação, um termo de *sort* '`Module`', do módulo nomeado pelo seu primeiro argumento `q`, um *quoted-identifier* (identificador com aspas simples, ou simplesmente *qid* de aqui em diante) como '`'NAT`'. O segundo argumento instrui a '`upModule`' a produzir (ou não) um metamódulo *achatado*: '`true`' significa criar a versão achatada, onde todos os módulos dependentes são incluídos na metarrepresentação. Esta função é parcial, já que o dado módulo pode não existir no banco de dados de módulos lidos por Maude.

$$\texttt{upModule} : \texttt{Qid} \times \texttt{Bool} \rightharpoonup \texttt{Module}$$

Como exemplo, o fragmento abaixo mostra a metarrepresentação do módulo interno '`TRUTH-VALUE`'.

```
Maude> red in META-LEVEL : upModule('TRUTH-VALUE, false) .
reduce in META-LEVEL : upModule('TRUTH-VALUE, false) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result FModule: fmod 'TRUTH-VALUE is
  nil
  sorts 'Bool .
  none
  op 'false : nil -> 'Bool [ctor special(...)] .
  op 'true : nil -> 'Bool [ctor special(...)] .
  none
  none
endfm
```

A função '`upTerm(u)`' converte um termo `u` da sua forma objeto (representado aqui por um argumento do *sort* interno '`Universal`') para sua metarrepresentação.

$$\texttt{upTerm} : \texttt{Universal} \rightarrow \texttt{Term}$$

Por exemplo, '`upTerm(true)`' produz a metarrepresentação da constante '`true`'.

```
Maude> red in META-LEVEL : upTerm(true) .
reduce in META-LEVEL : upTerm(true) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Constant: 'true.Bool
```

O exemplo acima envolvendo o módulo '`TRUTH-VALUE`' mostra que a metarrepresentação de módulos em Maude é bem parecida com a notação do nível objeto. Vamos discutir rapidamente a notaçaõ para metatermos. *Constantes* são representadas por um *qid* que consiste do nome da constante, um ponto e o nome do *sort*. Por exemplo, a metarrepresentação de '`true`', como mostramos anteriormente é ''`true.Bool`'. A metarrepresentação de variáveis segue a mesma linha, mas usa dois-pontos para separar o nome da variávei do nome do *sort*, tal como ''`b:Bool`'. Operadores que não são constantes são representados pelo nome do operador, incluindo os *underscores* usados na notação *mixfix*, com os argumentos dentro de colchetes. Por exemplo, num módulo que define o *sort* '`Foo`', com constantes '`a`' e '`b`', e uma operação binária '`_._`', a metarrepresentação de '`a . b`' é ''`_._['a.Foo,'b.Foo]`'.

Funções que movem do meta-nível para o nível objeto são as *funções descententes* (*down functions*). Até a versão 2.1.1, Maude não implementa o contraparte de '`upModule`', ou seja, uma função que move um metamódulo para um módulo. O reverso de '`upTerm`' é a função que move da metarrepresentação de um termo para sua forma objeto, '`downTerm`'. A operação recebe como primeiro argumento o metatermo e segundo argumento um termo que agirá como um "termo inválido" caso a conversão falhe.

$$\texttt{downTerm} : \texttt{Term} \times \texttt{Universal} \rightarrow \texttt{Universal}$$

Como exemplo, vamos converter de volta de ''`true.Bool`' para o nível objeto com a chamada '`downTerm('true.Bool, error-bool)`'. Aqui '`error-bool`' é uma constante préviamente criada que será produzida como resultado caso a conversão seja mal-sucedida. O resultado é, como esperado, '`true`', um termo com *sort* '`Bool`'. Se tentarmos converter um termo inválido, como ''`a.Bool`', o interpretador gerará um aviso e o termo resultante será '`error-bool`', as follows:

```
reduce in BOOL-META : downTerm('a.Bool, error-bool) .
Advisory: could not find a constant a of sort Bool in
          meta-module BOOL-TEST.
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: error-bool
```

Iremos concluir esta descrição com quatro funções que aumento as capacidades de metaprogramação de Maude. a função '`metaParse`' constrói um metatermo a partir de uma seqüência de *qids* passada como segundo argumento, usando a assinatura definida

pelo módulo passado como primeiro argumento. O tipo do termo sendo analisado é dado pelo terceiro argumento, ou '`anyType`', se o tipo não é conhecido préviamente. A função é parcial e resulta no termo de *sort* '`ResultPair?`' que contém ou uma tupla com o metatermo e seu tipo, ou uma mensagem de erro.

$$\texttt{metaParse} : \texttt{Module} \times \texttt{QidList} \times \texttt{Type?} \rightharpoonup \texttt{ResultPair?}$$

A seqüência de *qids* é usada, pois, como iremos ver na seção 2.3.4, o dispositivo '`LOOP-MODE`' em Maude converte toda entrada de texto feita pelo usuário entre parênteses numa seqüência de *qids*. Por exemplo, a entrada '`(mod F is sort A . endm)`', será convertida pelo '`LOOP-MODE`' em '`'mod 'F 'is 'sort 'A '. 'endm`'.

Como um exemplo do uso de '`metaParse`', vamos usar a assinatura definida pelo módulo interno '`BOOL`' para interpretar a frase '`true and false`'. Devemos passar esta frase para a '`metaParse`' como '`'true 'and 'false`'. O resultado é um '`ResultPair`' contendo o termo interpretado e o seu tipo, como se segue. Usamos a chamada '`upModule('BOOL, false)`' para obter a metarrepresentação do módulo interno '`BOOL`'.

```
reduce in META-LEVEL : metaParse(upModule('BOOL, false),
    'true 'and 'false, 'Bool) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'_and_['true.Bool,'false.Bool],'Bool}
```

Se passarmos uma entrada inválida para '`metaParse`', o resultado será '`noParse(n)`' onde `n` é a posição do *qid* problemático. No exemplo abaixo, a função não pôde entender o terceiro *qid*.

```
reduce in META-LEVEL : metaParse(upModule('BOOL, false),
    'true 'and '3, 'Bool) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair?: noParse(2)
```

O contraparte de '`metaParse`' é '`metaPrettyPrint`', que recebe uma assinatura, um termo, e produz a sua representação como uma seqüência de *qids*.

$$\texttt{metaPrettyPrint} : \texttt{Module} \times \texttt{Term} \rightarrow \texttt{QidList}$$

Ao passarmos o termo ‘`’_and_[’true.Bool,’false.Bool]`’ a esta função, junto com a assinatura definida no módulo ‘`BOOL`’, obtemos de volta ‘`’true ’and ’false`’.

```
reduce in META-LEVEL : metaPrettyPrint(upModule('BOOL, false),
    '_and_['true.Bool,'false.Bool]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result TypeList: 'true 'and 'false
```

(A razão pela qual Maude dá o *sort* de ‘`’true ’and ’false`’ como ‘`TypeList`’ é devido ao fato deste *sort* ser *subsort* de ‘`QidList`’, e, como mencionamos na seção 2.3, Maude sempre tentará imprimir menor *sort* aplicável a um termo.)

Para efetivamente executar módulos no metanível precisamos de funções que reescrevam e reduzam metatermos. A primeira é ‘`metaReduce`’, que recebe como entrada um metamódulo, um metatermo e produz um ‘`ResultPair`’ contendo o termo resultante da redução do metatermo no contexto do metamódulo.

$$\texttt{metaReduce} : \texttt{Module} \times \texttt{Term} \rightharpoonup \texttt{ResultPair}$$

A função ‘`metaRewrite`’ é o equivalente da ‘`metaReduce`’ para a reescrita de termos e necessita das mesmas opções de controle do comando ‘`rewrite`’. Recebe como argumentos um metamódulo, um metatermo, e um limite superior o número de reescritas. O resultado é também um ‘`ResultPair`’.

$$\texttt{metaRewrite} : \texttt{Module} \times \texttt{Term} \times \texttt{Bound} \rightharpoonup \texttt{ResultPair}$$

Vamos exemplificar a aplicação da função ‘`metaReduce`’. Considere o seguinte metamódulo, que contém duas constantes e uma função ‘`f`’, cujo valor é definido pela única equação presente.

```
fmod 'FOO is
  protecting 'BOOL .
  sorts 'Foo .
  none
  op 'a : nil -> 'Foo [none] .
  op 'b : nil -> 'Foo [none] .
  op 'f : 'Foo -> 'Foo [none] .
```

```
  none
  eq 'f['a.Foo] = 'b.Foo [none] .
endfm
```

O seguinte exemplo mostra uma execução da '`metaReduce`'.

```
reduce in META-LEVEL :
 metaReduce(fmod 'FOO ... endfm, 'f['a.Foo]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'b.Foo,'Foo}
```

O resultado da '`metaReduce`' é, como dito, um '`ResultPair`' com a metarrepresentação da constante '`b`' e seu *sort*, '`Foo`'.

A funcionalidade da '`metaRewrite`' é similar. Exemplificamos seu uso criando a metarrepresentação do módulo '`COUNTER`', usado como exemplo na seção 2.3.1. Os números estão na notação de Peano, com uma constante '`0`' e a função de sucessor '`s_`'. A notação '`s_^1000 (0)`'—metarrepresentada por '`s_^1000['0.Zero]`'—usa o fato do operador '`s_`' ter sido definido com o atributo '`iter`'. Com este atributo, `n` aplicações de um operador `f` a `x` podem ser escritas com a abreviação '`f^n (x)`'.

```
result SModule: mod 'COUNTER is
  protecting 'BOOL .
  protecting 'NAT .
  sorts 'Counter .
  none
  op 'counter : 'Nat -> 'Counter [none] .
  none
  none
  crl 'counter['n:Nat] => 'counter['_+_['n:Nat,'s_['0.Zero]]]
  if '_<_['n:Nat, 's_^1000['0.Zero]] = 'true.Bool [label('inc)] .
endm
```

Ao executarmos a função '`metaRewrite`' com um limite superior de vinte reescritas, temos o seguinte resultado, a metarrepresentação do termo '`counter(20)`' e seu *sort*, '`Counter`', como era de se esperar.

```
reduce in META-LEVEL : metaRewrite(mod 'COUNTER ... endm,
                                   'counter['0.Zero], 20) .
rewrites: 61 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'counter['s_^20['0.Zero]],'Counter}
```

2.3.2.1 Maude como uma metaferramenta

As capacidades reflexivas de Maude permitem o seu uso como uma *metaferramenta formal* [38, 39]. Para que uma ferramenta seja formal, ela precisa suportar uma *axiomatização precisa* da linguagem que está implementando. Isto é diferente de escrever ferramentas em linguagens convencionais como C ou Java, já que a implementação não é uma axiomatização formal.

O fato é que, como mencionamos na seção 2.2, a lógica de reescrita é um *framework lógico* que pode representar, de uma maneira *natural* diversas lógicas, linguagens, formalismos operacionais e modelos de computação. Esta representação natural é o resultado do uso de teorias MEL em conjunto com atributos equacionais (associatividade, comutatividade, identidade), possibilitando uma capacidade de representação bastante genérica com um cálculo dedutivo simples. Os aspectos formais são todos os aspectos lógicos de RWL e MEL discutidos na seção 2.2, em conjunto com sua executabilidade em Maude.

A representação de uma lógica $\mathcal{L}$ em lógica de reescrita é dada por um mapa de representação Ψ.

$$\Psi : \mathcal{L} \rightarrow \mathcal{R}$$

Combinando a sintaxe flexível dada por MEL em conjunto com as capacidades reflexivas descritas acima, este mapa pode ser implementado como uma função executável $\overline{\Phi}$ em Maude com a seguinte assinatura:

$$\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \rightarrow \texttt{Module}$$

em um módulo que estenda '`META-LEVEL`'. Aqui, $\texttt{Module}_{\mathcal{L}}$ é um tipo de dados abstrato que representa teorias na lógica $\mathcal{L}$. Ao usar as metafunções '`metaReduce`' e '`metaRewrite`' é possível *executar* $\mathcal{L}$ em Maude.

2.3.3 Dispositivos de entrada e saída

O mecanismo de entrada e saída em Maude é implementado pelo módulo '`LOOP-MODE`'. A construção básica do '`LOOP-MODE`' é o seu *objeto de loop*, do *sort* '`System`', cuja assinatura é a seguinte:

```
mod LOOP-MODE is
 protecting QID-LIST .

 sorts State System .

 op [_,_,_] : QidList State QidList ->
              System [ctor special (...)] .
endm
```

O objeto de *loop*, '`[_,_,_]`', contém três argumentos: o primeiro é o *fluxo de entrada*, que representa a entrada feita pelo usuário; o segundo é um termo abstrato de *sort* '`State`' que deve ser definido concretamente através de um *subsort* e que representa qualquer tipo de controle necessário para a aplicação; o terceiro é o *fluxo de saída*.

O objeto trabalha com um ciclo de leitura-execução-impressão da seguinte maneira: primeiro, um objeto de *loop* inicial é criado usando o operador '`[_,_,_]`' e o '`LOOP-MODE`' é iniciado passando-se este objeto inicial para o comando '`loop`'. Em seguida o usuário deve sempre digitar sua entrada entre parênteses para que seja redirecionada ao '`LOOP-MODE`'. Tudo o que for enviado desta maneira será convertido em *tokens* (representados por '`qids`') e posto no fluxo de entrada do objeto de *loop*. A partir deste ponto, o sistema deverá ter regras que casem com padrões específicos para lidarem com os diferentes comandos e declarações a serem feitos pelo usuário, possivelmente modificando o termo de *sort* '`State`'. A saída para o usuário deve ser feita colocando uma lista de *tokens* no fluxo de saída do objeto *loop*.

Vamos exemplificar criando um *loop* simples que aceita como entrada uma expressão aritmética e imprime de volta o seu valor computado. Começamos criando um objeto de *loop* inicial que saúda o usuário e pede por uma expressão. O estado deste objeto de *loop* guarda o valor da última expressão.

```
op last-value : Int -> State .
```

```
op init : -> System .
eq init = [ nil, last-value (0),
            'Olá! 'Por 'favor 'digite 'uma 'expressão. ] .
```

Começamos com uma regra simples: o comando '`(*)`' imprime o último valor computado. Na regra abaixo, '`QIL`' será casado com qualquer lista de *tokens* que representa o fluxo de saída e simplesmente adiciona a esta lista o que desejamos imprimir. A função interna '`string`' converte um número em uma *string* de acordo com a base desejada (10, neste caso). A função '`qid`' converte uma *string* em um '`qid`'.

```
rl [ '*, last-value (n), QIL ] =>
   [ nil, last-value (n), QIL
     'Último 'valor 'computado: qid(string(n,10)) ] .
```

Vamos definir a assinatura de nossa linguagem simples para expressões aritméticas, junto com um operador que calcula o seu valor. O módulo '`SIMPLE-LANGUAGE`' abaixo define dois *sorts*, '`Op`' e '`Exp`', que representam, respectivamente, as expressões e operações aritméticas na linguagem. Expressões são formadas pelo operador '`___`' que recebe duas expressões e um operador aritmético. Fazemos as expressões conterem inteiros criando a relação de *subsort* entre '`Int`' e '`Exp`'.

```
fmod SIMPLE-LANGUAGE is
 protecting INT .

 sort Op .
 ops plus minus times div : -> Op .
 sort Exp .
 subsort Int < Exp .
 op ___ : Exp Op Exp -> Exp .
```

O operador '`[[_]]`' abaixo recebe como entrada uma expressão e produz o seu valor, calculando recursivamente o valor de cada subexpressão. O final da recursão é o valor de um único inteiro.

```
 vars E1 E2 : Exp .
```

```
 var  I : Int .

 op [[_]] : Exp -> Int .
 eq [[E1 times E2]] = [[E1]] * [[E2]] .
 eq [[E1 plus  E2]] = [[E1]] + [[E2]] .
 eq [[E1 minus E2]] = [[E1]] - [[E2]] .
 eq [[E1 div   E2]] = [[E1]] quo [[E2]] .
 eq [[I]] = I .
endfm
```

Agora vamos mostrar as regras que de fato recebem uma expressão do usuário e imprimem ou o valor computado ou uma mensagem de erro. Inicialmente, precisamos analisar sintaticamente a entrada do usuário usando a assinatura criada no módulo '`SIMPLE-LANGUAGE`'. Se a análise sintática for bem sucedida, a regra procede da seguinta maneira: a entrada do usuário ('`QIL`') gerará um metatermo '`t`'; este metatermo será convertido de volta à representação objeto (uma expressão) pela '`downTerm`' e associado à metavariável '`exp`'; o operador '`[[_]]`' é então usado para obter o valor de '`exp`' e associá-lo à metavariável '`n'`'; a função '`metaPrettyPrint`' é usada para converter (a metarrepresentação de) '`n'`' para a lista de *tokens* que será exibida ao usuário. A regra ainda atualiza o estado '`last-value`' com $\mathtt{n}'$.

```
 crl [ QIL, last-value(n), QIL' ] =>
     [ nil, last-value (n'), QIL'
       'Resultado: metaPrettyPrint (SIMPLE-LANGUAGE, upTerm(n')) ]
 if QIL =/= nil /\
    t := getTerm (metaParse (SIMPLE-LANGUAGE, QIL, 'Exp)) /\
    exp := downTerm (t, error-exp) /\
    n' := [[exp]] .
```

E a análise sintática for mal sucedida, devemos gerar um "erro de sintaxe." A seguinte regra só será aplicada quando o resultado de '`metaParse`' não for do *sort* '`ResultPair`', o que representa a falha na análise sintática.

```
 crl [ QIL, last-value(n), QIL' ] =>
     [ nil, last-value (n), QIL' 'Erro 'de 'sintaxe! ]
 if QIL =/= nil /\
    not (metaParse (SIMPLE-LANGUAGE, QIL, 'Exp) :: ResultPair) .
```

Uma sessão completa do interpretador segue:

```
Olá! Por favor digite uma expressão.
Maude> (10 plus 30 minus 30)
Resultado: 10
Maude> (hello)
Erro de sintaxe!
Maude> (20 plus 30)
Resultado: 50
Maude> (50 times 200)
Resultado: 10000
Maude> (*)
Último valor computado: 10000
```

2.3.4 Full Maude

Full Maude é uma aplicação escrita em Maude que faz um grande uso da capacidade reflexiva da lógica de reescrita e Maude. Full Maude define uma álgebra de módulos rica que inclui hierarquia de módulos, parametrização, visões (*views*), teorias, expressões de módulos, e módulos orientados a objeto. Full Maude usa o material descrito nas seções 2.3.2 e 2.3.3 para implementar estas funcionalidades.

Esta seção descreve algumas das funcionalidades de Full Maude que são usadas no processo de conversão detalhado no capítulo 5. Full Maude usa o '`LOOP-MODE`' e toda a sua entrada deve ser feita entre parênteses.

2.3.4.1 Módulos de sistema e funcionais

Full Maude suporta módulos de sistema e funcionais. Todos os módulos lidos por Full Maude são armazenados em um *banco de dados de módulos* interno. Comandos como '`reduce`', '`rewrite`' e '`search`' também estão disponíveis. Isto é possível somente porque estes comandos também estão disponíveis no meta-nível como as funções '`metaReduce`', '`metaRewrite`' e '`metaSearch`'.

Como um exemplo introdutório, vamos mostrar uma sessão completa de Maude, começando pela leitura da apliação Full Maude, com um módulo '`FOO`' e um exemplo de reescrita.

```
                    \||||||||||||||||||/
                  --- Welcome to Maude ---
                    /||||||||||||||||||\
           Maude 2.1.1 built: Jun 15 2004 12:55:31
            Copyright 1997-2004 SRI International
                  Mon Apr 11 11:35:13 2005
Maude> load full-maude

              Full Maude 2.1.1 (July 20th, 2004)

Maude> (mod FOO is
         sort Foo .
         ops a b : -> Foo .
         rl a => b .
       endm)

rewrites: 717 in 119ms cpu (119ms real) (5975 rewrites/second)
Introduced module FOO

Maude> (rew a .)
rewrites: 123 in 6ms cpu (6ms real) (17576 rewrites/second)
rewrite in FOO :
  a
result Foo :
  b
```

2.3.4.2 Teorias, visões, e módulos parametrizados

Full Maude implementa módulos parametrizados e visões que habilitam uma modalidade de programação denominada de '`programação parametrizada`', seguindo a tradição iniciada pela linguagem OBJ [40]. Este tipo de programação envolve três elementos: os próprios *módulos parametrizados*; *teorias* que definem os requisitos de estrutura e propriedades de um parâmetro; e *visões* (*views*) que mapeiam uma teoria para um módulo correspondente. Em outras palavras, visões definem a interpretação dos parâmetros de um módulo parametrizado. Para simplificar a exposição, evitaremos descrever estes conceitos formalmente. Veja [40, 41] para mais informação.

Teorias são usadas para declarar interfaces de módulos e são criadas com a sintaxe '`fth` n `is` D `endfth`' para *teorias funcionais* e '`th` n `is` D `endth`' para *teorias de sistema*, onde n é o nome da teoria e D as suas declarações. A teoria mais básica, predefinida por Full Maude, é '`TRIV`' que define um único requisito: o *sort* '`Elt`'.

```
fth TRIV is
 sort Elt .
endfth
```

Teorias mais complexas podem ser criadas tal como a seguinte teoria de monóides, que requer uma constante qt1 e uma operação binária com identidade '`1`'.

```
fth MONOID is including TRIV .
 op 1 : -> Elt .
 op __ : Elt Elt -> Elt [assoc id: 1]
endth
```

A implementação de *Maude MSOS Tool* somente requer a teoria '`TRIV`'; evitaremos então discutir estas formas mais complexas e sua implicação na programação parametrizada de Full Maude. Veja [26, capítulo 15] para maiores detalhes.

Em relação a visões, iremos discutir também o mapeamento mais trivial, já que este é o único necessário na implementação de *Maude MSOS Tool*. Essencialmente, necessitamos de uma visão que mapeia o *sort* '`Elt`' para um *sort* concreto, definido em algum outro módulo. A sintaxe deste tipo de visão é:

```
view ν from TRIV to M is
 sort Elt to s .
endv
```

onde ν é o nome da visão, s um *sort* definido em M. É prática comum fazer o nome da visão ser o mesmo nome do *sort*. Por exemplo, a seguinte visão mapeia o *sort* '`Elt`' para o *sort* '`Exp`', definido no módulo '`EXP`'.

```
view Exp from TRIV to EXP is
 sort Elt to Exp .
endv
```

Módulos podem ser parametrizados por uma ou mais teorias. No caso da implementação de *Maude MSOS Tool*, apenas a teoria '`TRIV`' é usada como parâmetro. Um módulo parametrizado é declarado com a seguinte sintaxe no seu nome: '$\mathsf{M}(\mathsf{X}_1::\mathsf{T}_1 \mid \cdots \mid \mathsf{X}_n::\mathsf{T}_n)$', onde $\mathsf{X}_1, \ldots, \mathsf{X}_n$ são rótulos e $\mathsf{T}_1, \ldots, \mathsf{T}_n$ são teorias. Em Full Maude 2.1.1, todos os *sorts* vindos de teorias devem ser qualificados pelos seus rótulos. Se Z é o rótulo de um parâmetro, a teoria T, então todo *sort* s em T deve ser qualificado como '`Z@s`'. Como um exemplo disto, considere o seguinte módulo parametrizado que define conjuntos.

```
fmod SET(X :: TRIV) is
 sorts Set(X) .
 subsort X@Elt < Set(X) .

 op null : -> Set(X) .
 op __ : Set(X) Set(X) -> Set(X) [assoc comm id: null] .

 var E : X@Elt .

 eq E E = E .
endfm
```

O nome do módulo '`SET(X :: TRIV)`' indica que ele contém um único parâmetro, X, definido pela teoria '`TRIV`'; o *sort* '`Set(X)`' é um *sort parametrizado* que usa como parâmetro o rótulo X. Em geral, dado um módulo parametrizado '$\mathsf{M}(\mathsf{X}_1::\mathsf{T}_1 \mid \cdots \mid \mathsf{X}_n::\mathsf{T}_n)$', qualquer *sort* s pode ser escrito da forma '$\mathsf{s}(\mathsf{X}_1 \mid \cdots \mid \mathsf{X}_n)$'. Finalmente usamos '`X@Elt`' para referirmos ao *sort* '`Elt`', da teoria '`TRIV`'.

A instanciação de um módulo parametrizado é feita através da importação de uma *expressão de módulo* específica com a seguinte sintaxe: '$\mathsf{M}(\nu_1 \mid \cdots \mid \nu_n)$', onde M é um módulo parametrizado e $\nu_1, \ldots, \nu_n$ são as visões que mapeiam as teorias aos *sorts*. Por exemplo, ao definirmos uma visão de '`Elt`' para '`Int`', tal como:

```
view Int from TRIV to INT is
 sort Elt to Int .
endv
```

podemos usá-la para criar a expressão de módulo '`SET(Int)`' que instancia um módulo que define o conjunto de inteiros.

```
fmod TEST is
 including SET(Int) .
endfm
```

Após importarmos estes módulos, e visões para Full Maude, podemos usar um comando '`reduce`' como se segue:

```
Maude> (red 1 2 3 4 .)
reduce in TEST :
  1 2 3 4
result Set‘(Int‘) :
  1 2 3 4
```

2.3.4.3 Estendendo Full Maude

A entrada do usuário em Full Maude é gerenciada pelo módulo '`FULL-MAUDE`', que contém o *loop* principal usando o módulo '`LOOP-MODE`'. Este módulo contém um conjunto de regras de reescrita que casam com comandos específicos de Full Maude e chama as funções apropriadas para tratar aquela entrada. Lembre-se da seção 2.3.4 que toda entrada do usuário é convertida pelo '`LOOP-MODE`' numa lista de *tokens*. Ao passar esta lista, junto com a assinatura apropriada para a função '`metaParse`', obtemos uma metarrepresentação da entrada. Inicialmente, Full Maude analisa sintaticamente toda entrada de usuário usando o seu módulo '`GRAMMAR`'.

Concluída esta análise sintática, Full Maude deve chamar a função apropriada para lidar com o comando (ou módulo) declarado pelo usuário. Para que isto funcione, o módulo '`FULL-MAUDE`' inclui um conjunto de módulos, dentre os quais o '`DATABASE-HANDLING`'; Este módulo contém o pareamento entre os tipos de entrada que são entendidos por Full Maude e as funções que lidam com cada um eles.

O ciclo completo de leitura e execução da entrada do usuário é gerenciado através de um objeto do tipo '`Database`'. Este objeto contém diversos atributos, que incluem os *tokens* de entrada e saída, o nome do módulo corrente, o banco de dados de módulos que contém a metarrepresentação de todos os módulos que foram declarados pelo usuário. Este objeto tem a seguinte assinatura:

```
sort DatabaseClass .
```

```
subsort DatabaseClass < Cid .
op db :_ : Database -> Attribute .
op input :_ : TermList -> Attribute .
op output :_ : QidList -> Attribute .
op default :_ : ModName -> Attribute .
```

O módulo '`DATABASE-HANDLING`' define uma classe '`DatabaseClass`', *subsort* de '`Cid`', que é o *sort* que representa "identificadores de classes.". O atributo '`db`' contém o banco de dados de módulos, a principal construção de Full Maude. O atributo '`input`' contém a entrada do usuário processado pela '`metaParse`'. O atributo '`output`' contém uma lista de *tokens* ('`QidList`')—o que estiver neste atributo será enviado ao fluxo de saída e impresso no terminal do usuário. Finalmente o atributo '`default`' contém o módulo corrente que está sendo usado. Este último é necessário para que Full Maude saiba qual módulo deve ser usado quando um comando como '`reduce`', '`rewrite`' ou '`search`' for executado.

Para estender Full Maude com suporte uma nova linguagem $\mathcal{L}$ devemos criar um módulo funcional que contenha sua assinatura. Este módulo (digamos $\texttt{GRAMMAR}_{\mathcal{L}}$) define um tipo de dados $\mathsf{D}_{\mathcal{L}}$ que representa programas na linguagem $\mathcal{L}$. É desejável que esta extensão seja *conservativa*, ou seja, que a linguagem "original" de Full Maude ainda esteja disponível na versão estendida, com a mesma semântica. Isto é feito combinando ambas as assinaturas, `GRAMMAR` e $\texttt{GRAMMAR}_{\mathcal{L}}$, numa única, digamos `GRAMMAR+FM+L`. As regras para análise sintática da entrada do usuário devem ser modificadas para lidar com esta nova assinatura. Por exemplo, a seguinte regra recebe a entrada e a coloca no atributo '`input`' do objeto.

```
crl [in] :
  [QIL,
   < O : X@Database |
       db : DB, input : nilTermList, output : nil,
       default : MN, Atts >,
   QIL']
  => [nil,
      < O : X@Database | db : DB,
          input : getTerm(metaParse(GRAMMAR, QIL, 'Input)),
          output : nil, default : MN, Atts >,
       QIL']
  if QIL =/= nil /\
     metaParse(GRAMMAR, QIL, 'Input) : ResultPair .
```

Esta regra, '`[in]`', deve ser modificada para:

```
crl [in] :
  [QIL,
   < O : X@Database |
       db : DB, input : nilTermList, output : nil,
       default : MN, Atts >,
   QIL']
  => [nil,
      < O : X@Database | db : DB,
         input : getTerm(metaParse(GRAMMAR+FM+L, QIL, 'Input)),
         output : nil, default : MN, Atts >,
       QIL']
  if QIL =/= nil /\
     metaParse(GRAMMAR+FM+L, QIL, 'Input) : ResultPair .
```

Em seguida, o módulo '`DATABASE-HANDLING`' deve ser estendido com as funções definidas especificamente para lidar com os termos de $\mathrm{D}_{\mathcal{L}}$. Como exemplo, considere a seguinte regra em '`DATABASE-HANDLING`'. Ela "detecta" que o termo declarado pelo usuário e que foi corretamente analisado pela assinatura '`GRAMMAR`' casa com o padrão '`F[T, T']`' e que começa com '`fmod_is_endfm`' (módulo funcional), '`obj_is_endo`' (um outro nome para módulos funcionais para compatibilidade com a sintaxe de OBJ3), '`obj_is_jbo`' (outro

nome para módulos funcionais), '`mod_is_endm`' (módulo de sistema), or `omod_is_endom` (módulos orientados a objeto, não discutidos aqui). Ela prossegue então para chamar a função interna '`procUnit`' com o termo e o banco de dados como argumentos. Qualquer outra regra criada deve ser similar a esta.

```
crl [module] :
   < O : X@Database | db     : DB,  input   : (F[T, T']),
                      output : nil, default : MN, Atts >
   => < O : X@Database |
         db : procUnit(F[T, T'], DB), input : nilTermList,
         output :
           ('Introduced 'module
             modNameToQid(parseModName(T)) '\n),
         default : parseModName(T), Atts >
   if  (F == 'fmod_is_endfm) or-else
      ((F == 'obj_is_endo)   or-else
      ((F == 'obj_is_jbo)    or-else
      ((F == 'mod_is_endm)   or-else
       (F == 'omod_is_endom)))) .
```

O capítulo 5 descreve como Full Maude foi estendido para a implementação de *Maude MSOS Tool*.

2.4 Semântica de Reescrita Modular

Nesta seção iremos descrever brevemente a apresentação de Semântica de Reescrita Modular (*Modular Rewriting Semantics*, MRS) originalmente descrita em [25, 20].

A Semântica de Reescrita Modular [25, 20] é uma técnica para a especificação modular da semântica de linguagens de programação em lógica de reescrita. Uma especificação MRS é uma teoria de reescrita desenvolvida de acordo com algumas técnicas que suportam definições modulares. Especificações MRS usam um estilo de semântica orientado à sintaxe, com o texto do programa separado dos componentes semânticos, tais como o ambiente, a memória ou sinais de sincronização.

Vamos definir formalmente o aspecto de modularidade de uma especificação [25]. Quando $\mathcal{L}_1$ é uma extensão da linguagem $\mathcal{L}_0$, o primeiro requisito de modularidade é

monoticidade: existe uma inclusão de teorias $\mathcal{R}_{\mathcal{L}_0} \subseteq \mathcal{R}_{\mathcal{L}_1}$. Monoticidade informalmente significa que não há necessidade de *rever* as definições semânticas anteriores numa extensão subseqüente à linguagem. O segundo requisito de modularidade é de que termos instanciados sejam conservados (*ground conservativity*): para cada Σ_0-termo instanciado $t, t' \in T_{\Sigma_i}$ temos: (i) $E_0 \vdash t = t' \Leftrightarrow E_1 \vdash t = t'$, (ii) $\mathcal{R}_{\mathcal{L}_0} \vdash t \rightarrow t' \Leftrightarrow \mathcal{R}_{\mathcal{L}_1} \vdash t \rightarrow t'$. Este requisito informalmente significa que as definições semânticas não alteram a semântica das características originais dos fragmentos já definidos da linguagem.

MRS define então duas técnicas para a definição modular da semântica de linguagens de programação que satisfazem estes dois requisitos.

A primeira técnica de modularidade é a *herança de registros* (*record inheritance*), que é feita através do casamento de padrões *modulo* associatividade, comutatividade e identidade. Novos aspectos adicionados à linguagem podem levar à adição de novos componentes semânticos ao registro; no entanto, os axiomas dos aspectos antigos podem ser definitivos: eles também valem para o registro com novos componentes (repare que esta característica é compartilhada com MSOS: registros em MRS são, em essência, rótulos em em MSOS). A especificação em Maude da teoria equacional de registros (*records*) é a seguinte:

```
fmod RECORD is
 sorts Index Component Field PreRecord Record .
 subsort Field < PreRecord .

 op null : -> PreRecord [ctor] .
 op _,_ : PreRecord PreRecord
          ->  PreRecord [ctor assoc comm id: null] .
 op _=_ : [Index] [Component] -> Field [ctor] .
 op {_} : [PreRecord] -> [Record] [ctor] .
 op duplicated : [PreRecord] -> [Bool] .

 var I : Index .         vars C C' : Component .
 var PR : PreRecord .

 eq duplicated((I = C), (I = C'), PR) = true .
 cmb { PR } : Record if duplicated (PR) =/= true .
endfm
```

Um '`Field`' (campo) é definido como um par com projeções '`Index`' (índice) e um '`Component`' (componente); pares ilegais terão *kind* '`[Field]`'. Um '`PreRecord`' (pré-registro) é um conjunto com repetição (*multiset*) de campos, possivelmente vazio (representado pela constante '`null`'), formado pelo operador de união '`_,_`' declarado como associativo, comutativo e com identidade '`null`'. O axioma condicional de pertinência define um '`Record`' como um '`PreRecord`' "encapsulado" sem campos duplicados.

A técnica de herança de registros é baseada no fato de que podemos sempre considerar registros com mais campos como casos especiais de registros com menos campos. Por exemplo, um registro com um componente de ambiente ρ e um componente de memória σ pode ser visto como um caso particular do registro contendo apenas o componente ρ. Casamento de padrões *modulo* associatividade, comutatividade e identidade dá o suporte necessário à herança de registros porque sempre podemos usar uma variável extra '`PR`', do *sort* '`PreRecord`' para casar com qualquer campos extras que o registro possa ter. Por exemplo, a função '`get-env`' que extrai o componente de ambiente pode ser definida por '`eq get-env(env = E:Env, PR:PreRecord) = E .`' e irá ser aplicada ao registro onde os campos extras serão casados com '`PR`'.

A segunda técnica de modularidade é o uso sistemático de *interfaces abstratas*. Os *sorts* que especificam entidades sintáticas e semânticas são *sorts* abstratos de forma que: (i) especificam apenas *funções abstratas* que os manipulem; (ii) numa especificação, nenhum *sort* concreto sintático ou semântico é identificado com os *sorts* abstratos: eles são sempre especificados como *subsorts* de *sorts* abstratos correspondentes, ou são mapeados para *sorts* abstratos através de *coerções*; apenas no nível de *sorts* concretos que axiomas sobre funções abstratas são especificados.

O uso sistemático das duas técnicas acima parecem assegurar que a semântica de reescrita de uma extensão de linguagem $\mathcal{L}_0 \subseteq \mathcal{L}_1$ seja sempre modular, posto que: (i) as únicas regras de reescritas nas teorias $\mathcal{R}_{\mathcal{L}_0}$ e $\mathcal{R}_{\mathcal{L}_1}$ são regras semânticas

$$\langle \mathsf{f}(\mathsf{t}_1, \cdots, \mathsf{t}_n), \mathsf{u} \rangle \rightarrow \langle \mathsf{t}', \mathsf{u}' \rangle \Leftarrow \mathsf{C},$$

onde C é a condição da regra, f é um aspecto da linguagem, e.g., '`if-then-else`', u e u' são expressões de registros e u contém uma variável '`PR`' de *sort* '`PreRecord`'; (ii) a seguinte disciplina de *encapsulamento de informação* deve ser seguida em u, u', e qualquer expressão de registro que apareça em C: além da sintaxe do registro, apenas funções que apareçam na *interface abstrata* de algum dos campos do registro podem aparecer em expressões de registro; qualquer função auxiliar definida em *sorts* concretos

nos componentes dos campos nunca deve ser mencionada; e (iii) as regras semânticas para cada aspecto f da linguagem de programação devem ser definidas na *mesma* teoria, ou seja, ou todas estão em $\mathcal{R}_{\mathcal{L}_0}$ ou todas em $\mathcal{R}_{\mathcal{L}_1}$.

MRS usa pares, chamados de *configurações*; o primeiro componente é o *texto do programa* e o segundo é o *registro* cujos campos são os diferentes componente semânticos associados à computação do programa. Podemos especificar configurações em Maude com a seguinte teoria equacional:

```
fmod CONF
 is protecting RECORD .
 sorts Program Conf .
 op <_,_> : Program Record -> Conf [ctor] .
endfm
```

"Configurações restritas" são definidas pelo módulo '`RCONF`'. Elas são uma maneira de controlar as reescritas nas condições das regras MRS e são necessárias para a definição correta das transições da semântica operacional, como discutido na seção 2.4.1.

```
mod RCONF
 is extending CONF .

 op {_,_} : [Program] [Record] -> [Conf] [ctor] .
 op [_,_] : [Program] [Record] -> [Conf] [ctor] .

 vars P P' : Program .        vars R R' : Record .

 crl [step] : < P , R > => < P' , R' > if { P , R } => [ P' , R' ] .
endm
```

Para exemplificar iremos especificar a MRS de uma linguagem de programação bem simples que contém apenas uma construção '`let-in-end`' como em ML e a operação '`sum`' (adição). A mesma linguagem será usada na seção 4.5 para demonstrar o uso de *Maude MSOS Tool*. Começamos por definir sua sintaxe usando um módulo funcional '`SIMPLE-LANGUAGE-SYNTAX`'. Ele importa módulos '`EXP`' e '`ID`' que definem respectivamente, *sorts* '`Exp`' (as *expressões*) e '`Id`' (os identificadores). As duas construções da linguagem são definida pelos operadores '`let_=_in_end`' e '`_sum_`'.

```
fmod EXP is sort Exp . endfm
fmod ID  is sort Id .  endfm

fmod SIMPLE-LANGUAGE-SYNTAX is
 protecting INT .  protecting EXP .
 protecting ID .

 subsort Int < Exp .
 subsort Id < Exp .

 op let_=_in_end : Id Int Exp -> Exp .
 op _sum_ : Exp Exp -> Exp .
endfm
```

A specificação da construção '`let-in-end`' requer um ambiente de amarrações. Seguindo a técnica de modularidade de usar *funções abstratas* para definir componentes, definimos o módulo '`BASIC-ENVIRONMENT`' com o *sort* abstrato '`Env`', para ambiente. Além disto, as seguintes funções abstratas são definidas: '`find(ρ,i)`', que produz o valor amarrado ao identificador `i` no ambiente `ρ` e '`override(ρ,i,n)`', que sobrepõe o ambiente `ρ` com a amarração definida pelo identificador `i` e o inteiro `n`. '`Env`' é declarado como um *subsort* de '`Component`' e '`env`' uma constante do *sort* '`Index`'. Os axiomas de pertinência ao final do módulo definem o campo formado por '`env`' e termos do *sort* '`Env`'.

```
fmod BASIC-ENVIRONMENT is
 extending RECORD .  protecting INT .
 including ID .

 sorts Env .
 subsort Env < Component .

 op env : -> Index [ctor] .
 op find : Env Id -> [Int] .
 op override : Env Id Int -> Env .

 mb env = E:Env : Field .
endfm
```

O seguinte módulo, '`SIMPLE-LANGUAGE-SEMANTICS`', dá o significado para as construções '`let_=_in_end`' e '`_sum_`' usando uma semântica operacional *small-step* similar às regras descritas nas seções 4.5 e 2.1. Para que termos do *sort* '`Exp`' possam aparecer nas configurações, devemos fazer '`Exp`' *subsort* de '`Program`'. Em seguida, regras '`sum1`', '`sum2`' e '`sum3`' trabalham primeiro computando a expressão à esquerda de '`sum`', e depois à direita e, quando ambas expressões foram reduzidas a inteiros, simplesmente as reduz para a "soma de inteiros" dos valores. A regra '`let1`' computando a expressão '`E`' usando um ambiente sobreposto, através da função '`override`', com a amarração definida por '`X`' e '`I`'. Regra '`let2`' simplesmente nos informa que, quando a expressão chegar a um valor final, todo conjunto será substituído por esse valor. Finalmente, a regra '`find`' especifica que a computação de um identificador '`X`' é feita buscando o seu valor amarrado no ambiente '`Env`'

```
mod SIMPLE-LANGUAGE-SEMANTICS is protecting BASIC-ENVIRONMENT .
 protecting SIMPLE-LANGUAGE-SYNTAX .  protecting RCONF .

 subsort Exp < Program .

 vars E1 E2 E'1 E'2 E E' : Exp .  vars I1 I2 I I' : Int .
 vars Env Env' : Env .            vars PR PR' : PreRecord .
 var  X : Id .                    vars R R' : Record .

 crl [sum1] : { E1 sum E2, R } => [ E'1 sum E2, R' ]
 if { E1, R } => [ E'1, R' ] .
 crl [sum2] : { I sum E2, R } => [ I sum E'2, R' ]
 if { E2, R } => [ E'2, R' ] .
 crl [sum3] : { I1 sum I2, R } => [ I, R ]
 if I := I1 + I2 .
 crl [let1] : { let X = I in E end, { (env = Env), PR } }
     => [ let X = I in E' end, { (env = Env), PR' } ]
 if Env' := override (Env, X, I) /\ { E, { (env = Env'), PR } }
    => [ E', { (env = Env'), PR' } ] .
 rl [let2] : { let X = I in I' end, R } => [ I', R ] .
 crl [find] : { X, { (env = Env), PR } } => [ I, { (env = Env), PR } ]
 if I := find (Env, X) .
endm
```

Até o momento a especificação não é executável, já que devemos implementar as versões concretas das funções abstratas, definidas no módulo '`BASIC-ENVIRONMENT`'. Vamos mostrar uma possível implementação no módulo '`CONCRETE-ENVIRONMENT`' no qual as amarrações são construídas com o operador '`_|->_`' e o ambiente é o operador de justaposição associativo-commutativo de *sort* '`CEnv`'. A função de coerção '`<_>`' move os termos '`CEnv`' para '`Env`', desde que não haja amarrações duplicadas.

```
fmod CONCRETE-ENVIRONMENT is
 including BASIC-ENVIRONMENT .  protecting INT .

 sort Bind CEnv .
 subsort Bind < CEnv .

 op void : -> CEnv [ctor] .
 op _|->_ : Id Int -> Bind [ctor] .
 op __ : CEnv CEnv -> CEnv [ctor assoc comm id: void] .
 op <_> : [CEnv] -> [Env] .
 op dupl : [CEnv] -> [Bool] .

 var  X : Id .     vars I I' I1 : Int .
 var  CE : CEnv .

 eq dupl((X |-> I) (X |-> I1) CE) = true .
 cmb < CE > : Env if dupl(CE) =/= true .
 eq find(< (X |-> I) CE >, X) = I .
 eq override(< (X |-> I) CE >,X,I') =
    < (X |-> I') CE > .
 eq override(< CE >, X, I') =
    < (X |-> I') CE > [owise] .
endfm
```

O módulo '`SIMPLE-LANGUAGE`' reune todos os módulos numa única especificação.

```
mod SIMPLE-LANGUAGE is
 including SIMPLE-LANGUAGE-SYNTAX .
 including SIMPLE-LANGUAGE-SEMANTICS .
 including CONCRETE-ENVIRONMENT .
endm
```

Para executar um programa, criamos constantes '`x`' e '`y`', do *sort* '`Id`', e executamos com o comando '`rewrite`' de Maude a configuração com o programa e o registro contendo o ambiente vazio.

```
rewrite in SIMPLE-LANGUAGE :
  < let x = 10 in
     let y = 10 in x sum y end
    end,{env = < void >} > .
rewrites: 108 in 1ms cpu (10ms real) (108000 rewrites/second)
result Conf: < 20,{env = < void >} >

Bye.
```

2.4.1 Semântica de Reescrita Modular e MSOS

Semântica de Reescrita Modular tem uma relação próxima com a MSOS de Mosses. Isso se dá pelo fato de que SOS tem uma representação direta em lógica de reescrita [23, 18, 24, 20, 25] e que Semântica de Reescrita Modular e MSOS compartilham uma técnica de modularidade baseada no encapsulamento da informação semântica, chamada herança de registros em MRS. Em [42, 20] Braga e Meseguer propuseram uma transformação que preserva a semântica de MSOS para MRS com uma prova formal da bissimulação entre os modelos de MSOS e MRS.

O uso de expressões de rótulos em MSOS tais como $\{\rho = \rho_0, \tau' = \tau_0, \ldots\}$, como mencionamos na seção 2.1, é similar ao casamento de padrões de registros de Standard ML. Esta característica tem uma relação próxima ao uso da *herança de registros* em MRS onde a notação "`...`" é equivalente ao uso da metavaríavel '`PR`' de *sort* '`PreRecord`' que casa com qualquer componente não especificado.

Relembrando a forma geral de regras MSOS da seção 2.1:

$$\frac{c_1, \ldots, c_n}{c}$$

onde cada condição c_i é ou uma transição $v_i - \alpha_i \rightarrow v_i'$ um predicado p_i. A conclusão é uma transição $t - \alpha \rightarrow t'$, onde t, t', v_i, v_i' são ávores sintáticas com valores cujos *sorts* são *subsorts* de '`Program`'. Rótulos em MSOS são facilmente representados por registros do *sort* '`Record`' em Semântica de Reescrita Modular, usando uma metavariável do *sort* '`PreRecord`' para representar componentes não especificados no rótulo. Também precisamos de um *sort* '`IRecord`' para representar *rótulos não-observáveis* e '`IPreRecord`' para componentes não especificados de rótulos não observáveis, onde '`IRecord`' < '`Record`' e '`IPreRecord`' < '`PreRecord`', além de um operador de composição parcial '`_;_`' sobre '`Record`'. No restante desta seção, iremos usar as metavariáveis X, X' do *sort* '`Record`', metavariáveis U, U' do *sort* '`IRecord`', metavariáveis PR, PR' do *sort* '`PreRecord`', e metavariáveis UPR, UPR' do *sort* '`IPreRecord`'.

Além disto, é necessário entender a semântica das transições nas condições de regras MSOS. Ao contrário de lógica de reescrita, transições nas condições em MSOS são *transições de um passo*, enquanto que, em lógica de reescrita, são *transições de zero ou mais passos* devido ao cálculo dedutivo descrito na seção 2.2. De forma a conseguirmos simular corretamente as condições MSOS, precisamos de uma forma de controlar as reescritas nas condições. Uma forma de se fazer isto é usar a "configuração restrita," como a definida pelo módulo '`RCONF`' na seção 2.4.

Uma especificação MSOS $\mathcal{M}$ também define a sintaxe abstrata de uma linguagem de programação $\mathcal{L}$. Iremos assumir, nesta seção, que esta sintaxe, e qualquer componente semântico necessário são especificados como descrito na seção 2.4, ou seja, usando uma teoria de lógica equacional de pertinência $(\Omega_{\mathcal{L}}, E_{\mathcal{L}})$. Assumimos ainda que as transições MSOS têm uma notação representável (através da definição de uma *formal norma*, veja abaixo) definida por uma assinatura $(\Omega_{\mathcal{M}}, E_{\mathcal{M}})$. Seja (Ω, E) uma teoria em lógica equacional de pertinência que inclua ambas $(\Omega_{\mathcal{L}}, E_{\mathcal{L}})$ e $(\Omega_{\mathcal{M}}, E_{\mathcal{M}})$. Uma especificação MSOS $\mathcal{M}$ é uma teoria de lógica de reescrita $\mathcal{R} = (\Omega, E, R)$ onde cada regra de transição é representada por uma regra de reescrita em R. A transição de MSOS para MRS é, então, um mapa entre estas teorias.

$$\tau : (\Omega, E, R) \rightarrow (\Omega', E', R')$$

onde (Ω', E', R') inclui os módulos '`RECORD`' (com os *sorts* adicionais '`IRecord`' e '`IPreRecord`') e '`RCONF`' e as regras de transição R' são obtidas a partir de R como se segue. Primeiro, vamos assumir que as regras em R estão numa *forma normal*, ou seja:

- predicados presentes nas condições não envolvem registros, campos ou expressões de índices em seus argumentos;
- uma expressão de registro que apareça nas condições ou na conclusão é: (i) ou variáveis X ou U; (ii) um termo construtor com a forma geral $\{i_1 = w_1, \ldots, i_n = w_n, PR\}$ ou $\{i_1 = w_1, \ldots, i_n = w_n, UPR\}$, com $n \geq 0$, onde índices i_i são constantes do *sort* '`Index`' e podem ser primalizados e w_i são componentes cujo *sort* é um *subsort* de '`Component`' e existe uma assertiva de pertinência $i_i = w_i : \texttt{Field}$.

Como exemplo, considere a regra 2.13, na seção 2.1. Aquela regra pode ser reescrita da seguinte forma:

$$\frac{\alpha = \{env = \rho[m/x], \ldots\} \quad e_1 \dashrightarrow e_1'}{\texttt{let x=m in } e_1 \texttt{ end} \dashrightarrow \{env = \rho, \ldots\} \texttt{ let x=m in } e_1' \texttt{ end}} \tag{2.16}$$

A regra 2.16 não está na forma normal, posto que contém uma condição que envolve um registro. A regra 2.13 está na forma normal desejada. Mesmo não existindo um prova que o requisito de transições estarem numa forma normal não é restritivo, nossa experiência mostra que, de fato, não é.

O mapeamento τ é definido como se segue: (de [20]):

- (Ω', E') é obtido através de (Ω, E) da seguinte forma:
  - omitindo todos os índices primalizados e as suas equações e axiomas de pertinência relacionados, e adicionado um índice não-primalizado de cada índice *write-only*;
  - definindo *subsorts* '`ROPreRecord`', '`RWPreRecord`' e '`WOPreRecord`' (todos contendo a constante '`null`') do *sort* '`PreRecord`', correspondentes às partes do registro envolvendo campos *read-only*, leitura-escrita e *write-only*. Iremos usar as metavariáveis A, B e C sobre estes *sorts*, respectivamente, com suas variantes primalizadas A', B', C';

  - em '`WOPreRecord`' axiomatizamos um predicado de prefixo $\sqsubseteq$, onde $\mathsf{C} \sqsubseteq \mathsf{C}'$ significa que, para cada campo de *write-only* k a *string* $\mathsf{C.k}$ é um prefixo da *string* $\mathsf{C}'.\mathsf{k}$;
  - adicionando a assinatura do módulo '`RCONF`'

- R' contém a regra '`step`' em '`RCONF`', e para cada regra MSOS em R na forma normal, a seguinte regra de reescrita da forma é obtida:

$$\{\mathsf{t}, \mathsf{u}^{\mathrm{pre}}\} \to [\mathsf{t}', \mathsf{u}^{\mathrm{post}}] \Leftarrow \{\mathsf{v}_1, \mathsf{u}_1^{\mathrm{pre}}\} \to [\mathsf{v}_1', \mathsf{u}_1^{\mathrm{post}}] \wedge \cdots \cdots \wedge \{\mathsf{v}_n, \mathsf{u}_n^{\mathrm{pre}}\} \to [\mathsf{v}_n', \mathsf{u}_n^{\mathrm{post}}]$$

  onde para cada expressão u de registro na regra MSOS, $\mathsf{u}^{\mathrm{pre}}$ são $\mathsf{u}^{\mathrm{post}}$ definidos como se segue:

  - para uma expressão u de registro da forma X ou $\{\mathsf{PR}\}$, $\mathsf{u}^{\mathrm{pre}}$ é uma expressão de registro da forma $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ e $\mathsf{u}^{\mathrm{post}} = \{\mathsf{A}, \mathsf{B}', \mathsf{C}'\}$;
  - para uma expressão u de registro da forma U ou $\{\mathsf{UPR}\}$, $\mathsf{u}^{\mathrm{pre}}$ é uma expressão de registro da forma R ou $\{\mathsf{PR}\}$ e $\mathsf{u}^{\mathrm{post}} = \mathsf{u}^{\mathrm{pre}}$;
  - para uma expressão u de registro da forma $\{\mathsf{i}_1 = w_1, \ldots, \mathsf{i}_n = w_n, \mathsf{PR}\}$, com $n \geq 1$, $\mathsf{u}^{\mathrm{pre}}$ e $\mathsf{u}^{\mathrm{post}}$ são expressões de registro similares a u onde: (i) is uma expressão de um campo *read-only* $\mathsf{i} = w$ aparece em u, então ele aparece em $\mathsf{u}^{\mathrm{pre}}$ e $\mathsf{u}^{\mathrm{post}}$; (ii) se uma expressão de um campo *write-only* $\mathsf{i}' = w$ aparece em u, então $\mathsf{u}^{\mathrm{pre}}$ contém uma expressão de campo da forma $\mathsf{i} = \mathsf{l}$, sendo l uma nova variável de lista do tipo de dados correspondente, e $\mathsf{u}^{\mathrm{post}}$ contém uma expressão de campo da forma $\mathsf{i} = \mathsf{l} \cdot w$ (se u é o rótulo de uma condição, $\mathsf{u}^{\mathrm{pre}}$ contém $\mathsf{i} = \varepsilon$, onde ε é a identifdade do tipo de dados da lista, e $\mathsf{u}^{\mathrm{post}}$ contém $\mathsf{i} = w0$; (iii) se um par de expressões de campos *read-write* $\mathsf{i} = w$, $\mathsf{i}' = w'$ aparece em u então $\mathsf{u}^{\mathrm{pre}}$ contém $\mathsf{i} = w$ e $\mathsf{u}^{\mathrm{post}}$ contém $\mathsf{i} = w'$; (iv) PR é mapeada em $\mathsf{u}^{\mathrm{pre}}$ como $\mathsf{A}, \mathsf{B}, \mathsf{C}$ e em $\mathsf{u}^{\mathrm{post}}$ como $\mathsf{A}, \mathsf{B}', \mathsf{C}'$;
  - para uma expressão u de registro da forma $\{\mathsf{i}_1 = w_1, \ldots, \mathsf{i}_n = w_n, \mathsf{UPR}\}$, com $n \geq 1$, então os casos (i)–(iii) também se aplicam aqui e (iv) UPR é mapeado em ambos $\mathsf{u}^{\mathrm{pre}}$ e $\mathsf{u}^{\mathrm{post}}$ como PR;

- quaisquer condições que não são transições são mapeadas inalteradas;

- a condição é aumentada com $\mathsf{C} \sqsubseteq \mathsf{C}'$ se expressões da forma $\mathsf{A}, \mathsf{B}, \mathsf{C}$ e $\mathsf{A}, \mathsf{B}', \mathsf{C}'$ foram introduzidas nos termos $\mathsf{u}^{\mathrm{pre}}$ e $\mathsf{u}^{\mathrm{post}}$ na conclusão.

Como exemplo, a regra 2.13 é mapeada na seguinte:

$$
\begin{aligned}
&\{\ \texttt{let}\ x = m\ \texttt{in}\ e_1\ \texttt{end}, \{env = \rho, PR\}\} \\
&\rightarrow [\ \texttt{let}\ x = m\ \texttt{in}\ e_1'\ \texttt{end}, \{env = \rho, PR\}] \\
&\Leftarrow \{e_1, \{env = \rho[m/x], PR\}\} \rightarrow [e_1', \{env = \rho[m/x], PR\}]
\end{aligned}
$$

Capítulo 3

Trabalhos relacionados

Apesar de podermos especificar a semântica de uma linguagem de programação no estilo operacional usando linguagens e ferramentas formais de uso geral, optamos por descrever aqui ferramentas que foram desenvolvidas especificamente para especificações em semântica operacional. Desta forma, optamos por analisar neste capítulo LETOS [28] de Hartel, RML [43] de Pettersson e MSOS Tool [10] de Mosses, como três exemplos significativos de ferramentas de semântica operacional.

Começamos por descrever LETOS, "*A Lightweight Execution Tool for Operational Semantics.*" A ferramenta foi escrita usando uma combinação de C, lex e yacc e usa um dialeto que engloba a linguagem Miranda [44] para especificar a semântica operacional e denotacional, que é convertido em *scripts* Miranda (o autor menciona que, com pequenas mudanças, a saída pode ser modificada para Haskell [45]). Um aspecto adicional de LETOS é o seu suporte para a composição tipográfica de especificações em LaTeX (*pretty-printing*) e a capacidade de gerar traços de execução usando páginas HTML. LETOS tem um suporte parcial para especificações não-determinísticas, simulando-as usando funções que produzem listas e cujo resultado final será sempre um dos vários possíveis valores finais. Sintaxe abstrata é especificada usando Miranda e a sua forma de especificar tipos de dados definidos pelo usuário. Como é usual e especificações operacionais [7], LETOS permite a definição de diferentes relações entre configurações.

Vamos exemplificar estas características usando um exemplo simples, extraído da semântica da linguagem Mini-Freja [43]. Uma especificação LETOS é na verdade um documento LaTeX no qual a semântica é separada do restante do texto por marcadores '`.MS`' e '`.ME`'. O seguinte fragmento especifica a sintaxe abstrata de expressões em Mini-Freja. Apesar da sintaxe abstrata usar construções prefixadas, a ferramenta também permite construções infixadas.

```
\[
\begin{array}{@{}l@{}ll}
.MS
macro_Exp ::=
    CONSTexp macro_Const |
    VARexp macro_Var |
    CONSexp (macro_Exp, macro_Exp) |
    LAMexp (macro_Var, macro_Exp) |
    PRIMONEexp (macro_Primone, macro_Exp) |
    PRIMTWOexp (macro_Primtwo, macro_Exp, macro_Exp) |
    IFexp (macro_Exp, macro_Exp, macro_Exp) |
    APPexp (macro_Exp, macro_Exp) |
    CASEexp (macro_Exp, [macro_Rule]) |
    RECexp ([(macro_Var, macro_Exp)], macro_Exp) ;
.ME
\end{array}
\]
```

O fragmento acima é tipografado no seguinte:

$$
\begin{array}{ll}
\text{Exp}\equiv & \text{CONSTexp Const} \mid \\
& \text{VARexp Var} \mid \\
& \text{CONSexp(Exp, Exp)} \mid \\
& \text{LAMexp(Var, Exp)} \mid \\
& \text{PRIM1exp(Prim1, Exp)} \mid \\
& \text{PRIM2exp(Prim2, Exp, Exp)} \mid \\
& \text{IFexp(Exp, Exp, Exp)} \mid \\
& \text{APPexp(Exp, Exp)} \mid \\
& \text{CASEexp(Exp, [Rule])} \mid \\
& \text{RECexp([(Var, Exp)], Exp);}
\end{array}
$$

O seguinte fragmento (já tipografado) especifica uma relação, '`ifchoose`', que selecciona uma expressão baseado no valor do seu primeiro argumento. Ela é usada na semântica da construção "if" de Mini-Freja que mostramos em seguida.

$$\overset{\text{ifchoose}}{\Rightarrow} \quad :: \quad ((\mathsf{bool},\ \mathrm{Exp},\ \mathrm{Exp}) \leftrightarrow \mathrm{Exp});$$

$$[\mathsf{ifchoose}^{\mathsf{true}}] \quad (\mathsf{True},\ \mathsf{e2},\ \mathsf{e3}) \overset{\text{ifchoose}}{\Rightarrow} \mathsf{e2};$$

$$[\mathsf{ifchoose}^{\mathsf{false}}] \quad (\mathsf{False},\ \mathsf{e2},\ \mathsf{e3}) \overset{\text{ifchoose}}{\Rightarrow} \mathsf{e3};$$

Como exemplo do uso de diferentes relações numa transição, considere a semântica da construção "if" de Mini-Freja, dado por uma relação `eval` que usa a relação `ifchoose`.

$$\overset{\text{eval}}{\Rightarrow} \quad :: \quad ((\mathrm{Env},\ \mathrm{Exp}) \leftrightarrow \mathrm{Val});$$

$$[\mathsf{eval}^{\mathsf{if}}] \quad \frac{\begin{array}{l}(\mathsf{rho},\ \mathsf{e1}) \overset{\text{eval}}{\Rightarrow} \mathrm{CONSTval}(\mathrm{BOOLcnst}\ \mathsf{flag}),\\ (\mathsf{flag},\ \mathsf{e2},\ \mathsf{e3}) \overset{\text{ifchoose}}{\Rightarrow} \mathsf{e},\\ (\mathsf{rho},\ \mathsf{e}) \overset{\text{eval}}{\Rightarrow} \mathsf{v}\end{array}}{(\mathsf{rho},\ \mathrm{IFexp}(\mathsf{e1},\ \mathsf{e2},\ \mathsf{e3})) \overset{\text{eval}}{\Rightarrow} \mathsf{v};}$$

A próxima ferramenta analisada é RML (*Relational Meta Language*), que compila especificações em semântica natural [46]. O sistema consiste de um compilador escrito em Standard ML que compila RML em C de baixo nível. Como LETOS, RML permite a criação de diferentes relações para serem usadas em transições. RML não dá suporte para *pretty-printing* ou geração do traço de de especificações. Como um exemplo de especificação RML, vamos mostrar o fragmento equivalente de Mini-Freja do que foi exibido acima. Começamos pela definição da sintaxe abstrata de expressções. RML não permite a definição de funções infix, então a sintaxe é dada um estilo prefixado apenas.

```
datatype exp = CONSTexp of const
             | VARexp of var
             | CONSexp of exp * exp
             | LAMexp of var * exp
             | PRIM1exp of prim1 * exp
             | PRIM2exp of prim2 * exp * exp
             | IFexp of exp * exp * exp
             | APPexp of exp * exp
             | CASEexp of exp * crule list
             | RECexp of (var * exp) list * exp
```

A relação '`if_choose`' abaixo tem o mesmo significado do que a relação `ifchoose` de LETOS.

```
relation if_choose: (bool, Absyn.exp, Absyn.exp) => Absyn.exp =
 axiom if_choose(true, e2, _) => e2
 axiom if_choose(false, _, e3) => e3
end
```

Em seguida, a regra de transição da construção "if."

```
rule eval(rho, e1) => CONSTval(Absyn.BOOLcnst flag) &
     if_choose(flag, e2, e3) => e &
     eval(rho, e) => v
     ----------------
     eval(rho, Absyn.IFexp(e1,e2,e3)) => v
```

A última ferramenta analisada é MSOS Tool, escrita em Prolog, que converte especificações MSOS para Prolog. A MSOS Tool suporta os estilos *big-step* e *small-step* de especificação operacional e a sua linguagem de entrada é MSDF (*Modular SOS Definition Formalism*). A linguagem descrita no capítulo 4 é na verdade baseada na linguagem MSDF do MSOS Tool. A versão do MSOS Tool descrita aqui é baseada em [10].

Diferentemente de LETOS e RML, MSDF não permite a definição de diferentes relações. Apenas duas relações pré-definidas estão disponíveis: '`--->`', usada em semântica dinâmica em *small-step* e '`===>`', usada na semântica dinâmica em *big-step* e também na semântica estática. A sintaxe abstrata é especificada com notação parecida com a BNF, simlar a LETOS, mas apenas funções prefixadas podem ser definidas. O MSOS Tool está fortemente associado ao estilo *Constructive MSOS* (veja seção 6.1) de especificação de linguagens de programação e a ferramenta vem com uma grande biblioteca de módulos reutilizáveis que dão a semântica de várias construções básicas. As notas de aulas de Mosses ([10]) definem o mapeamento da sintaxe concreta para a abstrata de linguagens como bc e ML usando o mecanismo de *Definite Clause Grammars* (DCG) de Prolog.

Demonstraremos MSOS Tool usando como exemplo um dos módulos presentes na biblioteca fornecida com a ferramenta. Cada módulo é especificado usando três arquivos diferentes, organizados em uma estrutura de diretórios. O diretório '`Cons`' contém a sintaxe abstrata e a semântica de várias construções básicas de linguagens de programação. Dentro deste diretório '`Cmd`' contém as construções relacionadas ao aspecto imperativo de linguagens de programação. Selecionamos o diretório '`cond-nz`' que contém um comando

condicional. Começamos apresentando a sintaxe abstrata, dada num arquivo chamado ‘ABS.msdf’. Dependências com outros módulos são resolvidas automaticamente pela ferramenta: no exemplo, o usuário não precisa se preocupar em ter que incluir os módulos que definem os conjuntos ‘Cmd’ e ‘Exp’.

```
Cmd ::= cond-nz(Exp,Cmd)
```

A semântica estática para função é especificada no arquivo ‘CHK.msdf’:

```
ValueType ::= int

  E ===> int, C ===> void
--------------------------
cond-nz(E,C):Cmd ===> void
```

A semântica dinâmica é especificada no arquivo ‘RUN.msdf’:

```
Value ::= Integer

               E --{...}-> E'
---------------------------------------
cond-nz(E,C):Cmd --{...}-> cond-nz(E',C)

cond-nz(0,C):Cmd ---> skip

        N \= 0
----------------------
cond-nz(N,C):Cmd ---> C
```

A semântica de diversas linguagens de programação é dada em exemplos no diretório ‘Lang’. Por exemplo, o seguinte código Prolog demonstra o uso de DCG para definir a tradução de bc para ‘cond-nz’.

```
stmt(cond_nz(E,C)) -->
        "if", i, "(", i, expr(E), i, ")", i, stmt(C).
```

MSOS Tool não dá suporte para *pretty-printing* de especificações ou geração do traço de programas.

Capítulo 4

Maude MSOS Tool

Este capítulo descreve *Maude MSOS Tool* (MMT), um ambiente de programação para especificações MSOS. MMT é uma ferramenta formal no sentido descrito em [38], implementada como uma extensão conservativa de *Full Maude*, que compila especificações MSOS em lógica de reescrita. A implementação é baseada no mapeamento inicialmente proposto em [18] e posteriormente revisto em [25]. A compilação em lógica de reescrita possibilita o uso das ferramentas formais presentes em Maude para a verificação e execução de especificações MSOS.

A sintaxe dos módulos aceita por MMT é baseada no *Modular SOS Definition Formalism* (MSDF) descrito por Mosses em [10]. Denominamos a linguagem usada por MMT também de MSDF pois é idêntica em muitos aspectos à MSDF de Mosses. As pequenas diferenças, descritas no capítulo 7 que existem são causadas por idiosincrazias do analisador sintático de Maude.

MMT foi desenvolvido com o objetivo primário de ser um ambiente formal para a especificação linguagens de programação; contudo, semântica operacional (e MSOS) é aplicável a uma grande variedade de tópicos. Por exemplo, semântica operacional foi usada no desenvolvimento de vários cálculos de concorrência, como CCS [4] e o π-*calculus* [5]. Contudo, dado o seu proprósito principal, optamos por mostrar o uso das diferentes construções MSDF motivando o seu uso na semântica formal de uma linguagem de programação $\mathtt{L}$.

Dar a semântica de $\mathtt{L}$ é, de fato, dar a semântica de cada uma de suas construções $\mathtt{L_c}$. Consiste na descrição formal da *sintaxe* e *semântica* de $\mathtt{L_c}$. Em relação à sintaxe, seguimos a estratégia tradicional de evitar as complicações da sintaxe concreta adotando uma versão abstrata de $\mathtt{L_c}$ (veja, por exemplo, [1, Seção 1.4.1] e [10, Seção 1.2.3] para uma discussão sobre este assunto). Vamos denominar esta construção abstrata de $\mathcal{L}_\mathtt{c}$.

Também seguindo as práticas usuais, descrevemos a gramática livre de contexto de $\mathcal{L}_{\mathtt{c}}$ usando a notação Backus Naur Form (BNF).[1] A especificação da sintaxe abstrata de uma linguagem de programação é o assunto da seção 4.2.2.

A semântica da construção $\mathcal{L}_{\mathtt{c}}$ é definida por *regras de transição*, com *rótulos* contendo os componentes semânticos necessários, como discutido na seção 2.1. Descrevemos como rótulos e transições são especificados em MSDF nas seções 4.2.3 e 4.2.4 respectivamente.

Especificações são organizadas em módulos para permitir a construção modular de uma especificação, algo desejável não apenas para propósitos didáticos, mas também como um princípio saudável de engenharia para lidar com a complexidade inerente em projeto de larga escala. O uso de módulos é discutido na seção 4.2.1.

4.1 Notação

Nas seções que seguem, a gramática das construções MSDF é especificada usando uma notação BNF estendida, com terminais em '`fonte teletipo e entre aspas imples`' e não-terminais especificados em ⟨ fonte sem serifa e entre colchetes angulares ⟩. A notação BNF estendida é sempre usada numa expressão entre metaparênteses: '(E)*' e '(E)$^+$' denotam, respectivamente, zero ou mais, e uma ou mais, repetições de E; '(E)$^?$' denota uma expressão opcional.

Antes de explicarmos as construções MSDF primeiro devemos discutir alguns aspectos léxicos. Sendo uma extensão de *Full Maude*, MMT também faz uso do dispositivo '`LOOP-MODE`' de Maude (seção 2.3), e está restrito à sua capacidade de análise sintática. Para isolarmos esta descrição dos detalhes técnicos da análise sintática de Maude, usamos alguns não-terminais pré-definidos de uma forma abstrata: o primeiro é ⟨ id ⟩, de *identificadores*, que segue a definição de Maude para identificadores. Até a versão 2.1.1, as regras léxicas para identificadores são como se segue ([26]):

> Qualquer *string* finita de caracteres ASCII tal que:
>
> - não contenha nenhum espaço em branco;
> - os caracteres '`{`', '`}`', '`(`', '`)`', '`[`', '`]`' e '`,`' quebram uma seqüência de caracteres em vários identificadores;

[1] Uma perspectiva histórica interessante no nome Backus Naur Form *versus* Backus Normal Form é discutida em [47].

- o caractere do sinal de crase ‘‘’ é usado como um caractere de escape para indicar que um espaço em branco ou um caractere especial que o precede não quebra a seqüência.

Identificadores com a letra inicial em caixa alta são representados por ⟨ upper-id ⟩, enquanto aqueles cuja letra inicial é em caixa baixa são representados por ⟨ lower-id ⟩. Com a versão 2.1.1 não é possível especificar este e outros requisitos—por exemplo, que um identificador consista apenas de letra—usualmente definidos através de expressões regulares. MMT os checa depois que a análise léxica termina.

Outro não-terminal pré-definido é ⟨ term ⟩ que representa ávores sintáticas com valores que apareçam em transições.

Em Full Maude, toda entrada do usuário deve ser feita entre parênteses.

4.2 Sintaxe de MSDF

Estas seções descrevem os elementos da linguagem de especificação MSDF. Cada seção começa com uma exposição formal das construções, seguida de um exemplo ilustrativo e termina com detalhes no seu uso.

4.2.1 Módulos

⟨ module ⟩ → ‘`msos`’ ⟨ id ⟩ ‘`is`’ (⟨ see ⟩)* (⟨ declaration ⟩)* ‘`sosm`’
⟨ see ⟩ → ‘`see`’ ⟨ name ⟩ (‘`,`’ ⟨ name ⟩)* ‘`.`’

Todos os módulos em MSDF começam com ‘`msos`’ e terminam com ‘`sosm`’. Um nome de módulo é um identificador (⟨ id ⟩). Para incluir outros módulos, deve-se usar a construção ⟨ see ⟩, que é a palavra ‘`see`’, seguida por uma lista de nomes de módulos separados por vírgulas; várias linhas de importação são permitidas para uma maior flexibilidade, mas ela são equivalentes a uma única linha de importações.

O restante do módulo consiste numa seqüência de declarações MSDF, representada pelo não-terminal ⟨ declaration ⟩. Para permitir uma maior flexibilidade na declaração, a ordem das declarações não importa, já que todas as declarações são primeiramente coletadas antes que a compilação começe.

Como exemplo de inclusão de módulos, considere o seguinte módulo MSDF, que define um módulo '`PL-SYNTAX`' que, por sua vez, inclui os módulos que definem os diversos componentes de uma linguagem de programação hipotética PL.

```
msos PL-SYNTAX is
 see PL-EXPRESSIONS, PL-DECLARATIONS .
 see PL-IMPERATIVES, PL-ABSTRACTIONS .
 see PL-CONCURRENCY .
sosm
```

De um modo geral, para ser usado num módulo, uma declaração MSDF precisa ser definida previamente em outro módulo e este ser explicitamente incluído. Contudo, algumas inclusões podem ser omitidas não apenas para abreviar as especificações mas também aumentar a clareza das mesmas. Não é incomum em definições de linguagens de programação um conjunto básico de construções ser usado e quase todos os módulos subseqüentes, já que construções primitivas, como comandos e expressões, são usadas até mesmo nos aspectos mais avançados; seria entediante e poderia levar a erros forçar o usuário a explicitar a inclusão destes módulos em cada módulo onde são necessários, especialmente onde eles são *obviamente* necessários.

O conceito de "necessidade óbvia" é subjetivo e, para evitar confusão, o MMT é uma regra simples para omitir uma inclusão de módulo: todos os módulos que definam conjuntos que são usados em um módulo são, por *default*, incluídos. Isto inclui não apenas conjuntos usados na definição de tipos de dados mas também usados na definição do rótulo.

Por exemplo, considere o seguinte módulo MSDF que define uma sintaxe abstrata para expressões condicionais.

```
msos COND is
 Exp ::= if Exp then Exp else Exp .
sosm
```

MMT automaticamente incluirá o módulo que contenha a definição do conjunto '`Exp`'. Se este módulo for, digamos, '`EXP`', então o módulo expandido, sem nenhuma inclusão implícita, seria:

```
msos COND is
```

```
 see EXP .

 Exp ::= if Exp then Exp else Exp .
sosm
```

É importante enfatizar que esta regra é aplicável apenas a *conjuntos*; se um módulo define uma construção que usa outras construções definidas externamente, os módulos necessários devem ser explicitamente incluídos. Considere o seguinte exemplo, onde um '`while`' é definido em termos de condicionais ('`if then else`') e seqüenciamento de comandos ('`;`'). Os módulos que definem estas construções ('`COND`' e '`SEQ`', respectivamente) tiveram que ser explicitamente incluídos, enquanto que os módulos que definem os conjuntos '`Exp`' e '`Cmd`' foram omitidos.

```
msos WHILE is
  see COND, SEQ .

  Cmd ::= while Exp do Cmd .

  (while Exp do Cmd) : Cmd -->
   if Exp then (Cmd ; while Exp do Cmd) else skip .
sosm
```

4.2.2 Definições de tipos de dados

⟨ setid ⟩ → ⟨ upper-id ⟩
⟨ opid ⟩ → ⟨ lower-id ⟩
⟨ dsetid ⟩ → ⟨ setid ⟩ | ⟨ setid ⟩'*' | ⟨ setid ⟩'+'
⟨ mixfix-fun ⟩ → ⟨ opid ⟩ | ⟨ dsetid ⟩
⟨ dec ⟩ → ⟨ setid ⟩ '.' | ⟨ setid ⟩ ':=' ⟨ dec-rhs ⟩ ('|' ⟨ dec-rhs ⟩)* '.'
⟨ dec-rhs ⟩ → ⟨ dsetid ⟩ | ⟨ mixfix-fun ⟩$^{+}$ ('['⟨ attr ⟩']')$^{?}$
⟨ param ⟩ → ⟨ dsetid ⟩ | '('⟨ dsetid ⟩')List' | '('⟨ dsetid ⟩')Set'
    | '('⟨ dsetid ⟩','⟨ dsetid ⟩')Map'
⟨ equiv ⟩ → ⟨ setid ⟩ '=' ⟨ param ⟩ '.'
⟨ declaration ⟩ → ⟨ dec ⟩ | ⟨ equiv ⟩

Apesar de especificada em notação BNF, a sintaxe abstrata de uma linguagem de programação em MSDF é na verdade uma definição algébrica de tipos de dados. Isto também é verdade em outras ferramentas para semântica operacional, como LETOS (de Hartel) e MSOS Tool (de Mosses). Neste sentido, não-terminais são conjuntos e seqüências de não-terminais e terminais ($\langle$ mixfix-fun $\rangle^+$) são funções n-árias especificadas na chamada forma *mixfix* onde os não-terminais representam os argumentos e os terminais representam o nome da função no formato *mixfix*. A função que contenha um único terminal é chamada de *constante*.

Em MSDF, um $\langle$ setid $\rangle$ é chamado de *conjunto primitivo*, um identificador cuja primeira letra deve estar em caixa alta ($\langle$ upper-id $\rangle$), tal como '`Integer`'. Este identificador deve conter apenas letras, devido a restrições relacionadas à formação de metavariáveis (seção 4.2.4); um $\langle$ dsetid $\rangle$ é um *conjunto derivado*, já que representa um conjunto automaticamente derivado dos conjuntos primitivos. Existem dois tipos de conjuntos derivados: o conjunto das seqüências de elementos e o conjunto das seqüências não-vazias de elementos. Para um conjunto `s`, o primeiro é escrito como `s*` e o segundo como `s+`.

O não-terminal $\langle$ dec $\rangle$ declara ou um novo conjunto, ou uma inclusão de conjuntos ou uma declaração de função, dependendo do seu lado direito ($\langle$ dec-rhs $\rangle$). Se o lado direito é um único $\langle$ dsetid $\rangle$, então é uma inclusão de conjuntos (e.g., '`Exp ::= Value .`'). Se o lado direito é uma seqüência de identificadores, então é interpretado como a declaração de uma função *mixfix*, como descrito acima (e.g., '`Exp ::= if Exp then Exp else Exp .`'). Uma forma prefixada é um caso particular de a forma geral *mixfix* (e.g., '`Sys ::= parallel (Cmd, Cmd)`'). Os atributos opcionais que podem ser especificados para uma função (o não-terminal '`attr`' entre colchetes) são discutidos ao final desta seção.

Atualmente, MMT define préviamente dois conjuntos: inteiros ('`Int`') e booleanos ('`Boolean`'), e suas respectivas seqüências '`Int*`', '`Int+`', '`Boolean*`' e '`Boolean+`'. O conjunto '`Boolean`' contém duas constantes, '`tt`' e '`ff`', que representam os valores verdadeiro e falso, respectivamente.

Conjuntos parametrizados, definidos pelo não-terminal '`param`', são obtidos usando-se um modificador sobre um ou dois conjuntos: dado um conjunto `s`, '`(s)List`' é o conjunto de listas de elementos de `s`; '`(s)Set`' é o conjunto de conjunto finitos de elementos de `s`; dado um segundo conjunto `k`, '`(s,k)Map`' é o conjunto de mapeamento finitos de elementos de `s` para `k`. Conjuntos parametrizados não devem ser usados diretamente, e sim através dos seus *conjuntos equivalentes*, usando-se a sintaxe do não-terminal '`equiv`'.

O seguinte fragmento exemplifica o uso das definições de tipos de dados. Quatro conjuntos são declarados: '`Exp`', '`Value`', '`Cmd`' e '`Id`'. Estes conjuntos representam, respectivamente, as expressões, valores, comandos e identificadores de uma linguagem de programação qualquer. Em seguida uma inclusão de conjuntos é definida entre '`Value`', e os conjuntos pré-definidos '`Int`' e '`Boolean`'. Em seguida as sintaxes abstratas de duas construções são definidas na forma *mixfix*: uma expressão condicional, e um comando de *loop*, que espera receber uma seqüência não-vazia de comandos como seu corpo. Finalmente, uma equivalência entre o conjunto '`Env`' e '`(Id, Value)Map`'—um mapa de identificadores para valores—é definido.

```
Exp .  Value .
Cmd .  Id .
Value ::= Int | Boolean .
Exp ::= if Exp then Exp else Exp .
Cmd ::= while Exp do Cmd+ .
Env = (Id, Value)Map .
```

Finalmente funções em MSDF podem ter *atributos* para simplificar a sua notação. Eles são, atualmente, os mesmos suportados por Maude: '`associative`' (abreviado '`assoc`'), usado para define uma função binária associativa; '`commutative`' ('`comm`'), usado para definir uma função binária comutativa; '`precedence n`' ('`prec n`'), usado para dar a precedência a uma função, onde `n` é um natural entre zero e $2^{31} - 1$ (valores menores têm preferência a valores maiores na análise léxica); e '`identity:t`' ('`id:t`') que estabelece o termo `t` como a identidade da função especificada. Como em Maude, estes atributos podem ser combinados.

Atributos em MSDF permite a criação de uma sintaxe abstrata mais flexível, ao mesmo tempo que mantêm a notação simples. O uso de valores de precedência, por exemplo, permite a especificação de funções aritméticas com o grupamento correto na sintaxe abstrata, o que normalmente não acontece (o grupamento normalmente acontece na sintaxe concreta e movido explicitamente para a abstrata). O documento de especificação de Java, emph-Java Language Specification [48], por exemplo, define não-terminais '`UnaryExpression`', '`MultiplicativeExpression`', '`AdditiveExpression`' e '`RelationalExpression`' para lidar com os diferentes níveis de precedencia destes diferentes expressões. Usando MSDF, pode-se especificar os mesmos requisitos usando atributos de precedência e associatividade, como se segue.

```
Exp .
Exp ::= Exp + Exp [assoc prec 10] .
Exp ::= Exp - Exp [assoc prec 30] .
Exp ::= Exp * Exp [assoc prec 20] .
Exp ::= Exp / Exp [assoc prec 20] .
Exp ::= Exp < Exp [assoc prec 10] .
Exp ::= Exp > Exp [assoc prec 10] .
Exp ::= Exp == Exp [assoc prec 10] .
```

Ao pedirmos para Maude analisar sintaticamente a expressão '`e1 + e2 * e3 * e4 * e5 - e6`' usando as declarações acima, obtemos o seguinte termo, exibido com parênteses para explicitar a ordem de agrupamento: '`(((e1 + e2) * (e3 * (e4 * e5))) - e6)`'.

4.2.3 Rótulos

⟨ label ⟩ → '`Label={`' ⟨ field ⟩ ('`,`' ⟨ field ⟩)* '`, ...} .`'
⟨ field ⟩ → ⟨ index ⟩ '`:`' ⟨ derived ⟩
⟨ index ⟩ → ⟨ lower-id ⟩
⟨ declaration ⟩ → ⟨ label ⟩

Cada módulo MSDF pode conter no máximo uma declaração de rótulo usando a sintaxe do não-terminal ⟨ label ⟩. Uma declaração de rótulo consiste numa seqüência de declarações de tipos de campos (⟨ field ⟩). Cada campo consiste num índice (⟨ index ⟩), e o tipo de seu componente (⟨ derived ⟩).

Os índices dos componentes que definem um campo podem ser *read-only*, *read-write* ou *write-only*: se existe um único índice não primalizado, então o campo define um componente *write-only*, como em:

```
Label = { env : Env, ... }
```

Um índice que aparece primalizado não primalizado define um component *read-write*. Ambos componentes devem ter o mesmo tipo.

```
Label = { st : Store, st' : Store, ... }
```

Finalmente, um único índice primalizado define um componente *write-only*. O único tipo admissível de componentes *read-only* são seqüências de conjuntos primitivos, como em:

```
Label = { output' : Value*, ... }
```

Se exitem várias declarações de rótulos um único módulo, apenas a última é considerada. Uma única declaração pode definir vários campos:

```
Label = { env : Env, st : Store, st' : Store,
          output' : Value*, ... }
```

4.2.4 Transições semânticas

⟨ transition ⟩ → ⟨ cond-transition ⟩ | ⟨ uncond-transition ⟩
⟨ cond-transition ⟩ → ⟨ cond ⟩ (‘,’ ⟨ cond ⟩)* (‘[’ ⟨ label ⟩ ‘]’)?

quad ‘--’ ⟨ step ⟩ ‘.’
⟨ uncond-transition ⟩ → (‘[’ ⟨ label ⟩ ‘]’)? ⟨ step ⟩ ‘.’
⟨ label ⟩ → ⟨ id ⟩
⟨ typed-term ⟩ → ⟨ term ⟩ ‘:’ ⟨ dsetid ⟩
⟨ step ⟩ → ⟨ typed-term ⟩ ⟨ relation ⟩ ⟨ term ⟩
⟨ cond-step ⟩ → ⟨ term ⟩ ⟨ relation ⟩ ⟨ term ⟩
⟨ relation ⟩ → ‘-->’ | ‘==>’ | ‘-’ ⟨ label-exp ⟩ ‘->’ | ‘=’ ⟨ label-exp ⟩ ‘=>’
⟨ label-exp ⟩ → ‘{’ ⟨ field-exp ⟩ (‘,’ ⟨ field-exp ⟩)* ‘}’
⟨ field-exp ⟩ → ⟨ index ⟩ ‘=’ ⟨ term ⟩ | ⟨ composition ⟩ | ⟨ rest ⟩
⟨ rest ⟩ → ‘...’ | ‘-’ | ‘X’ | ‘U’
⟨ composition ⟩ → ⟨ rest ⟩ ‘;’ ⟨ rest ⟩
⟨ cond ⟩ → ⟨ eq ⟩ | ⟨ pred ⟩ | ⟨ inst ⟩ | ⟨ cond-step ⟩
⟨ eq ⟩ → ⟨ term ⟩ ‘=’ ⟨ term ⟩
⟨ pred ⟩ → ⟨ term ⟩
⟨ inst ⟩ → ⟨ term ⟩ ‘:=’ ⟨ term ⟩
⟨ declaration ⟩ → ⟨ transition ⟩

O não-terminal ⟨ transition ⟩ especifica como transições MSOS são escritas em MSDF. Vamos começar descrevendo transições incondicionais (⟨ uncond-trans ⟩), compostas pelo não-terminal ⟨ step ⟩, com um rótulo opcional ⟨ label ⟩.[2] Uma transição incondicional estabelece três possíveis relações entre termos: semântica “big-step” (‘==>’), semântica “small-step” (‘-->’), e semântica estática (também ‘==>’), seguindo a notação tradicional

[2]Este rótulo é apenas decorativo e não deve ser confundido com o rótulo da transição MSOS.

da literatura que usa semântica operacional [10, 1, 7]. Todas as três relações, descritas pelo não-terminal ⟨ relation ⟩, são ternárias com os seguintes componentes: a árvore sintática com valores tipada a ser casada, a expressão do rótulo MSOS (⟨ label-exp ⟩) envolta entre chaves e a árvore sintática com valores resultante.

Uma expressão de rótulo é uma lista não vazia de expressões de campos (⟨ field-exp ⟩) separadas por vírgulas e envolta entre chaves, onde cada expressão de campo é ou um índice e seu componente ou o "resto do rótulo." Existem metavariáveis especiais para o resto do rótulo (⟨ rest ⟩): se é não-observável, a metavariável '`-`' deve ser usada, caso contrário '`...`' deve ser usada. Como é comum em especificações MSOS, usamos `-->` e `==>` como alternativas para `-{-}->` e `={-}=>`, respectivamente. Ao invés de '`...`' e '`-`', pode-se usar as metavariáveis '`X`' e '`U`', respectivamente. Estas últimas podem opcionalmente conter um número ao final, como '`X1`'.

Composição de rótulos é especificada usando a sintaxe do não-terminal ⟨ composition ⟩. Recomenda-se que metavariáveis numeradas sejam usadas na composição ('`X1`', '`X2`', etc.), seguindo a notação tradicional de MSOS. Composição de rótulos apenas faz sentido em transições condicionais.

Metavariáveis sobre conjuntos não são declaradas explicitamente em MSDF, e sim implicitamente: todos os não-terminais da forma ⟨ dsetid ⟩ que apareçam nas transições são consideradas metavariáveis para os seus conjuntos correspondentes usando uma regra simples de formação: dado que todos os identificadores de conjuntos devem ter apenas letras (seção 4.2.2), todos os caracteres que não são nem letras nem os símbolos '`*`' ou '`+`' são removidos do ⟨ dsetid ⟩ e o restante é assumido ser o tipo da metavariável.

Por exemplo: '`Exp`', '`Exp1`' e '`Exp'`' são todas metavariáveis sobre o conjunto '`Exp`', obtido removendo-se os caracteres '`1`' e '`'`', respectivamente, enquanto '`Exp*1`' é uma metavariável sobre '`Exp*`', obtida removendo-se o '`1`'. A explicação para esta característica de MSDF é que, em definições de linguagens de programação, é comum o uso de um conjunto em quase todas as regras de transição. Por exemplo, o conjunto de expressões ou o conjunto de valores. Como foi no caso da inclusão de módulos, seria provavelmente entediante e passível de erros forçar o programador a declarar o mesmo conjunto de variáveis em todos os módulos repetidamente.

Como um exemplo de transições não condicionais, expressões de rótulos e metavariáveis implícitas, considere o seguinte fragmento que define a semântica de uma construção que imprime um valor computado. Ela define uma transição incondicional entre a árvore sintática com valores tipada '`(print Value) : Cmd`' e '`skip`', o comando "não

faça nada." A semântica do comando '`print`' é definida com um componente *write-only* pela expressão de rótulo '`out' = Value`' que modela a informação produzida '`Value`'. O resto do rótulo não é observável, ou seja, qualquer componente de *read-write* não é modificado e nenhuma outra informação é produzida por outros componentes de *write-only*; componentes de *read-only* não se modificam.

```
(print Value) : Cmd -{out' = Value, -}-> skip .
```

Transições condicionais têm quatro diferentes tipos de condições, governadas pelo não-terminal: condições equacionais, predicados, instanciações de variáveis, e transições. Condições equacionais (⟨ eq ⟩) são satisfeitas verificando-se a igualdade de dois termos, tal como '`first (Pids') = Int`'. Um único termo '`P`' pode ser usado como predicado (⟨ pred ⟩), tal como '`odd(n)`', para abreviar a condição equacional '`P = true`'. O não-terminal ⟨ inst ⟩ define a sintaxe a ser usada na instanciação de novas metavariáveis. A metavariável livre deve estar no lado esquerdo do '`:=`', tal como em '`Value := lookup (Id, Env)`'. Finalmente, ⟨ cond-step ⟩ é uma transição que tem a mesma sintaxe das transições condicionais, com a exceção de que o tipo da árvore sintática a ser casado é necessariamente o *menor conjunto* (em relação à ordenação dos conjuntos) que se aplica ao lado esquerdo da transição.

Vamos exemplificar transições condicionais com uma especificação em *small-step* da semântica de uma construção condicional com sintaxe '`cond Exp Exp Exp`'. A semântica é a usual: a primeira expressão é computada até um valor booleano; se for verdadeiro, a segunda expressão é computada, caso contrário, a terceira é computada. A semântica para isto precisa de três transições: a primeira especifica que a condição—a metavariável '`Exp`' sobre o conjunto '`Exp`'—deve ser computada. O termo '`(cond Exp Exp1 Exp2) : Exp`' é uma árvore sintática tipada cujo tipo é '`Exp`', enquanto que na transição o tipo da árvore tipada '`Exp`' é implicitamente o menor conjunto aplicável, que no caso é também '`Exp`'. A condição diz que, se '`Exp`' é computada para '`Exp'`', então o lado esquerdo da transição é computado para '`cond Exp' Exp1 Exp`'. O rótulo '`...`' na condição é o mesmo na conclusão, significando que qualquer mudança em componentes de *read-write* é propagada para a transição, e qualquer informação produzida por componentes *write-only* é também produzida pela conclusão. Componentes de *read-only*, devem sempre permanecer inalderados.

```
                        Exp -{...}-> Exp'
-- ------------------------------------------------------
(cond Exp Exp1 Exp2) : Exp -{...}-> cond Exp' Exp1 Exp2 .
```

No exemplo acima, usamos o fato de que, em Maude, três traços ('`---`') iniciam um comentário de linha.

Finalmente, duas regras adicionais são necessárias para cada possível valor final de '`Exp`': se for '`tt`', todo o lado esquerdo é computado em '`Exp1`', caso contrário em '`Exp2`'. Estas transições não são observáveis.

```
(cond tt Exp1 Exp2) : Exp --> Exp1 .
(cond ff Exp1 Exp2) : Exp --> Exp2 .
```

Vamos exemplificar transições condicionais definido a mesma construção condicional no estilo *big-step* que também usa três regras, mas com uma construção adicional.

A construção interna (que não faz parte da sintaxe da linguagem) '`if-choose`' tem três argumentos: um valor e duas expressões. Se o valor for '`tt`', ela é computada em '`Exp2`', caso contrário em '`Exp3`'. O exemplo abaixo também mostra regras com rótulos decorativos.

```
Exp ::= if-choose (Value, Exp2, Exp3) .

[if-choose-tt] if-choose (tt, Exp2, Exp3) : Exp ==> Exp2 .
[if-choose-ff] if-choose (ff, Exp2, Exp3) : Exp ==> Exp3 .
```

É necessário uma única transição para a construção propriamente dita, mas com três transições nas condições. A primeira espera que expressão '`Exp`' compute em '`Value`', com expressão de rótulo '`X1`'. Este valor, quando usado como parâmetro em '`if-choose (Value, Exp2, Exp3)`', deve ser computado em '`Exp'`', de uma maneira não observável. Finalmente, '`Exp''`' é computado em '`Value''`', com expressão de rótulo '`X2`'. Se estas três condições forem satisfeitas e as expressões de rótulos '`X1`' e '`X2`' forem componíveis, então todo o lado esquerdo é computado em '`Value''`'. O rótulo '`X1 ; X2`' na conclusão especifica a composição dos rótulos '`X1`' e '`X2`'.

```
      Exp ={X1}=> Value,
         if-choose (Value, Exp1, Exp2) ==> Exp',
                                    Exp' ={X2}=> Value'
 [if] -- ------------------------------------------
      (cond Exp Exp1 Exp2) : Exp ={X1 ; X2}=> Value' .
```

MMT testa por varíaveis dependentes da fonte (*source-dependent variables*) (seção 2.1) e reporta quando encontra variáveis que não tenham esta propriedade, como o exemplo abaixo mostra:

```
(msos TEST is
 Foo .
 Bar .

 Foo ::= f (Bar, Bar) .

         Bar --> Bar'
 -- -------------------------
 f(Bar, Bar) : Foo --> Bar'' .
sosm)
```

Ao ler o módulo 'TEST', o seguinte erro é exibido.

```
rewrites: 19081 in 96ms cpu (101ms real) (196741 rewrites/second)
ERROR: non source-dependent variables found: Bar'' in module TEST
```

4.3 Operações pré-definidas sobre conjuntos derivados e parametrizados

Os conjuntos parametrizados e derivados em MSDF têm várias operações associadas, que descrevemos nesta seção. Cada função é apresentada de acordo com o esquema $\mathsf{f} : \mathsf{S} \rightarrow \mathsf{S}'$: uma função f com domínio S e imagem S'. Se a função é no formato *mixfix*, é convertida para o formato prefixado onde os argumentos são substituídos pelo carácter '_'.

Sejam s e s' dois conjuntos primitivos. Lembre-se que, de um conjunto s os conjuntos $\mathsf{s}*$ e $\mathsf{s}+$ são derivados automaticamente. Além disto o usuário pode criar os seguintes conjuntos parametrizados: `(s)List`, `(s)Set` e `(s, s')Map`.

4.3.1 Seqüências

`_,_` : s∗ × s∗ → s∗

A função binária da monóide (com parênteses opcionais), cuja identidade é '`()`'. Um único elemento `s` é também uma seqüência.

4.3.2 Listas

`[_]` : s∗ → `(s)List`

Constrói uma lista a partir de uma seqüência de elementos.

`_in_` : s × `(s)List` → `Bool`

Verifica se `s` pertence à lista `(s)List`.

`first` : `(s)List` → `s`

Primeiro elemento da lista `(s)List`.

`remove` : s × `(s)List` → `(s)List`

Cria uma cópia da lista `(s)List` com os elementos `s` removidos.

`insert-back` : s × `(s)List` → `(s)List`

Insere `s` no final de `(s)List`.

`insert-front` : s × `(s)List` → `(s)List`

Insere `s` no início de `(s)List`.

`length` : `(s)List` → `Nat`

Número de elementos de `(s)List`.

4.3.3 Mapas

`_|->_` : s × s′ → `(s,s′)Map`

Cria uma entrada no mapa que amarra `s` a s′.

`_+++_` : `(s,s′)Map` × `(s,s′)Map` → `(s,s′)Map`

União de mapas que não têm intersecção.

`length` : `(s,s′)Map` → `Nat`

Número de entradas no mapa.

`def lookup` : s × (s,s′)Map → Bool

Verifica se existe um mapeamento para s em (s,s′)Map.

`lookup` : s × (s,s′)Map → s′

Retorna o elemento mapeado por s em (s,s′)Map.

`_/_` : (s,s′)Map × (s,s′)Map → (s,s′)Map

Sobrepõe os mapeamentos do segundo (s,s′)Map com os do primeiro first (s,s′)Map.

4.3.4 Conjuntos

`size` : (s)Set → Nat

Número de elementos de (s)Set.

`_in_` : s × (s)Set → Bool

Verifica se (s)Set contém s.

`_+_` : (s)Set × (s)Set → (s)Set

União de conjuntos que não têm intersecção.

4.4 Interface com o usuário

A operação normal do MMT é com o usuário carregando módulos MSDF no prompt ou lendo arquivos contendo módulos MSDF usando o comando '`load`' de Maude, além de usar os comandos de Full Maude para reescrita, redução, busca e verificação de especificações. Esta seção exibe os comandos que são específicos ao MMT.

O único comando atualmente disponível controla o *step flag* do processo de compilação, detalhado no capítulo 5. A operação normal do MMT é com este *flag ativado*, dado que, como mostramos na seção 2.4, precisamos controlar a reescrita nas condições. O único caso onde este *flag* deve estar desligado é quando existe alguma outra forma de controlar estas reescritas, por exemplo, usando a linguagem de definição de estratégias para Maude de Alberto Verdejo [49]. A sintaxe do comando é '`step flag on`' ou '`step flag off`'.

4.5 Exemplo

Esta seção mostra um exemplo simples do uso de MMT baseado no exemplo inicialmente exibido na seção 2.4. Este exemplo, como visto, descreve uma linguagem bastante simples: apenas uma construção '`let-in-end`' e uma construção aritmética '`sum`'.

A especificação, de nome '`SIMPLE-LANGUAGE`', deve iniciar com '`msos`' e finalizar com '`sosm`', como mostra o exemplo abaixo:

```
(msos SIMPLE-LANGUAGE is ... sosm)
```

As seguintes definições de tipos de dados declaram os conjuntos usados na especificação: '`Exp`', o conjunto de expressões e '`Id`' o conjunto de identificadores. O conjunto de expressões engloba o conjunto de identificadores e inteiros (o conjunto prédefinido '`Int`').

```
Id .
Exp .
Exp ::= Int | Id .
```

Declaramos agora a sintaxe das duas construções da linguagem.

```
Exp ::= let Id = Int in Exp end
      | Exp sum Exp .
```

A especificação do '`let-in-end`' requer um componente de *read-only* para as amarrações. Declaramos como um mapa de identificadores para inteiros. Este ambiente é acessível pelo índice '`env`'.

```
Env = (Id, Int)Map .
Label = { env : Env, ... } .
```

Agora, para a semântica dinâmica. Para computadr a soma de duas expressões, primeiro computamos a primeira expressão até que atinja um valor final, que deve ser um inteiro nesta linguagem; em seguida, a segunda expressão é computada. Quando esta produz um valor final, o valor da construção é a soma matemática dos dois inteiros.

```
                Exp1 -{...}-> Exp'1
-- ---------------------------------------
(Exp1 sum Exp2) : Exp -{...}-> Exp'1 sum Exp2 .

                Exp2 -{...}-> Exp'2
-- ---------------------------------------
 (Int sum Exp2) : Exp -{...}-> Int sum Exp'2 .

       Int3 := Int1 + Int2
-- ----------------------------
(Int1 sum Int2) : Exp --> Int3 .
```

Para o significado da construção '`let-in-end`', a expressão é computada no contexto do ambiente corrente sobreposto pela amarração definida pela declaração '`Id = Int`'. Quando a computação atinge um valor final, a expressão inteira é reescrita para este valor.

```
  Env' := (Id |-> Int) / Env, Exp -{env = Env', ...}-> Exp'
-- ------------------------------------------------------
(let Id = Int in Exp end) : Exp -{env = Env, ...}->
                                 (let Id = Int in Exp' end) .

(let Id = Int in Int' end) : Exp --> Int' .
```

A computação de um identificador busca o valor amarrado a ele no ambiente.

```
   Int := lookup (Id, Env)
-- -------------------------
Id : Exp -{env = Env}-> Int .
```

Para rodar programas com esta especificação, devemos criar identificadores; fazemos isto criando as constantes '`x`' and '`y`'.

```
(msos TEST is
 see SIMPLE-LANGUAGE .
```

```
 Id ::= x | y .
sosm)
```

Podemos usar agora o comando '`rewrite`' de Full Maude para executar um programa simples, cujo argumento é uma configuração MRS (veja seção 2.4).

```
(rewrite < (let x = 10 in
             let y = 10 in
              x sum y
             end
            end) ::: 'Exp,
            { env = void } > .)

rewrite in TEST : < ... >
result Conf :
  < 20 ::: 'Exp, { env = void } >

Bye.
```

Capítulo 5

A implementação de MMT

Este capítulo descreve a implementação de MMT. Lembre-se da seção 2.3.2.1 em que a representação de uma linguagem/lógica $\mathcal{L}$ em lógica de reescrita é feita por um mapa $\Psi : \mathcal{L} \to \mathcal{R}$ que é, em última análise, expressado por uma função executável em Maude $\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \to \texttt{Module}$. No caso de MSOS, esta função é $\overline{\Phi} : \texttt{Module}_{\mathcal{M}} \to \texttt{Module}$, onde $\texttt{Module}_{\mathcal{M}}$ é um tipo de dados que representa módulos MSDF e `Module` o tipo de dados que representa os módulos de sistema em Full Maude.

Este capítulo está organizado da seguinte maneira: a seção 5.1 descreve a função inicialmente descrita na seção 2.3.4.3 aplicada ao caso de módulos MSDF; a seção 5.2 faz uma descrição de alto-nível do processo de compilação, descrevendo como módulos MSDF são compilados; a seção 5.3 descreve como a declaração de tipos de dados é convertida para Maude; a seção 5.4 descreve como a informação de rótulos é usada no processo de compilação; e finalmente a seção 5.5 descreve como transições MSDF são compiladas em regras de reescrita.

5.1 MMT como uma extensão de Full Maude

Dado que MMT deve ser uma extensão conservativa de Full Maude precisamos de conseguir analisar sintaticamente ambas as linguagens, MSDF e Full Maude. Ou seja, precisamos não somente criar a assinatura para as construções MSDF, mas também *combiná-la* com a assinatura de Full Maude para analisar sintaticamente a entrada do usuário. O módulo funcional '`MSOS-SL-SIGNATURE`' define a assintura das declarações MSDF e combinamos com o módulo '`GRAMMAR`' de Full Maude para criar o módulo '`MSOS+FM-GRAMMAR`'. Esta assinatura composta é usada nas regras do *loop* de comando para analisar a entrada do usuário e redirecioná-la às funções corretas. Para isto, precisamos substituir o módulo

'FULL-MAUDE' com um módulo próprio, chamado 'MAUDE-MSOS-TOOL', que contenha tais regras. Este módulo inclui o módulo 'EXT-DATABASE-HANDLING' que estende o módulo 'DATABASE-HANDLING' com as funções que se aplicam a módulos MSDF.

Vamos dar um exemplo concreto. Lembre-se da seção 2.3.4.3 como a entrada do usuário é processada em Full Maude. No caso do MMT, quando o usuário digita alguma coisa via 'LOOP-MODE', esta é convertida para 'QidList' e casado na regra '[in]' pela variável 'QIL'. A regra então tenta analisar sintaticamente esta lista de *tokens* com a assinatura composta e, caso seja bem sucedida, coloca o termo analisado no atributo 'input' do objeto de banco de dados.

```
crl [in] : [QIL, < O : X@Database |
                   db : DB, input : nilTermList, output : nil,
                   default : MN, Atts >, QIL'] =>
           [nil, < O : X@Database | db : DB,
                  input : getTerm(metaParse(MSOS+FM-GRAMMAR,
                                  QIL, 'Input)),
                  output : nil, default : MN, Atts >, QIL']
if QIL =/= nil /\
   metaParse(MSOS+FM-GRAMMAR, QIL, 'Input) : ResultPair .
```

O módulo 'EXT-DATABASE-HANDLING' define regras específicas para cada tipo de termo de entrada. A seguinte regra funciona da seguinte maneira: se o termo de entrada casa com o padrão ''msos_is_sosm[T, T']', que é um móduo MSDF como definido pelo módulo 'MSOS-SL-SIGNATURE', ele é passado como parâmetro para a função mmt-proc-unit que inicia o processo de compilação.

```
rl < O : X@Database | db : DB, input : ('msos_is_sosm[T, T']),
                      step-flag : B,
                      output : nil, default : MN, Atts > =>
   < O : X@Database | db : mmt-proc-unit('msos_is_sosm[T, T'],
                      step-flag(B), DB),
                      input : nilTermList, step-flag : B,
                      output : ('Introduced 'MSOS 'module
                             modNameToQid(parseModName(T)) '\n),
                      default : parseModName(T), Atts > .
```

Os passos completos são: a análise sintática da entrada '`QidList`' do usuário produz um termo do *sort* '`MSOSUserInput`' que é a metarrepresentação da entrada do usuário, definida pelo módulo '`MSOS-SL-SIGNATURE`'. Precisamos sair desta metarrepresentação para um termo de *sort* '`MSOSModule`' que é a metarrepresentação de módulos MSDF propriamente dita, como definida pelo módulo '`MSOS-UNIT`'. Este processo em duas fases—entrada de usuário para '`MSOSUserInput`' e então para '`MSOSModule`'—é usado porque um módulo MSDF contém uma *sintaxe definida pelo usuário* em que os termos que aparecem nas transições são definidos no *mesmo* módulo em que são usados. Por causa disto, no momento em que são lidos, não temos nenhuma informação sobre a assinatura definida pelo usuário. A solução adotada por Maude e seguida por MMT é usar *bubbles* (bolhas) no lugar dos termos. *Bubbles* são seqüência de *tokens* que ainda não foram analisados sintaticamente, pois não há assinatura para analisá-los ainda. Depois que a assinatura é coletada do módulo MSDF, ela então é usada para analisar as bolhas. Uma vez que todas as *bubbles* tenham sido analisadas, temos uma metarrepresentação *completa* de módulos MSDF, de acordo com a definição do módulo '`MSOS-UNIT`', e a compilação pode ser iniciada.

A primeira fase do processo de análise sintática produz um termo que é a metarrepresentação de um módulo '`MSOSUserInput`' module. Este não é o módulo MSDF, como dissemos, mas uma metarrepresentação do que foi digitado pelo usuário. Deste módulo, obtemos o módulo '`MSOSModule`', através da função '`mmt-proc-unit`', definida no módulo '`MSOS-PARSER`', com a seguinte assinatura:

$$\texttt{mmt-proc-unit} : \texttt{Term} \times \texttt{CFlags} \times \texttt{Database} \rightarrow \texttt{Database}$$

em que '`Term`' é o termo analisado, '`CFlags`' são os *flags* de compilação que controlam determinados aspectos da compilação, descritos na seção 5.2, e '`Database`' é o banco de dados de módulos. Em outras palavras, esta função recebe um termo e um banco de dados, e produz o banco de dados modificado inserindo o chamado "*term unit*" do módulo MSDF; lembre-se da seção 2.3.4 que o "*term unit*" é a metarrepresentação da entrada do usuário.

Após a inserção do *term unit* no banco de dados, '`mmt-proc-unit`' passa o controle para '`mmt-eval-preunit`', que deve construir uma assinatura a partir da informação de tipos de dados do módulo MSDF e analisar as *bubbles* nele. A função está definida no módulo '`MSOS-SOLVER`' e tem a seguinte assinatura:

`mmt-eval-preunit` : `Unit` × `CFlagsDatabase` → `Database`

Esta função espera receber uma '`Unit`', que é um *sort* maior do que o *sort* '`MSOSModule`'. Ela passa a resolver as *bubbles* chamando a função '`solve-module-bubbles`' do módulo '`MSOS-SOLVE-BUBBLES`'.

`solve-module-bubbles` : `Unit` × `Database` → `Unit`

Depois que as *bubbles* foram resolvidas, o controle retorna para a função '`mmt-eval-preunit`' que em seguida chama '`convertMSOS`', que é a função que de fato produz a teoria de reescrita associada com a especificação MSDF.

O restante deste capítulo é dedicado a descrever a funcionalidade desta função.

5.2 Módulos

Vamos começar apresentando uma descrição de alto-nível do processo de compilação. Dada uma especificação MSDF $\mathcal{M}$, cada *módulo MSDF* é convertido num *módulo de sistema* que usa as teorias de reescritas de MRS; informações dos tipos de dados de $\mathcal{M}$ são convertidas para axiomas em lógica equacional de pertinência e as relações ternárias (*big-step*, *small-step* e semântica estática) são convertidas para as relações duais definidas pelas regras de reescritas sobre configurações MRS; *expressões de rótulo* são convertidas em *expressões de registro* em MRS e devem ser divididas em duas partes: registros no lado esquerdo (*antes*) e no lado direito (*depois*) de regras MRS.

Para formalizar isto, vamos considerar um módulo MSDF abstrato $\mathcal{M}$ como uma tripla $(\mathsf{D}, \mathsf{L}, \mathsf{T})$, sendo D as declarações de tipos de dados, L as declarações de rótulos e T as regras de transição. O processo de compilação gera um módulo de sistema de Full Maude `Module` que contém a teoria $(\Sigma, \mathsf{E}, \mathsf{R})$, sendo Σ a assinatura, E o conjunto de equações e R o conjunto de regras de reescrita. A assinatura Σ contém o conjunto parcialmente ordenado $(\mathsf{S}, \subseteq)$ de *sorts* e conjunto O de operadores.

No MMT esta operação é feita pela função '`convertMSOS`', definida no módulo '`MSOS-CONVERTER`', com a seguinte assinatura:

`convertMSOS` : `MSOSModule` × `CFlags` × `Database` → `StrSModule`

em que '`MSOSModule`' é o tipo de dados de $\texttt{Module}_{\mathcal{M}}$ na implementação de MMT. A imagem de '`convertMSOS`' é '`StrSModule`', a metarrepresentação em Full Maude de módulos `Module`. O *sort* '`CFlags`' representam os *flags* que controlam o processo de compilaçaõ; apenas um é usado atualmente que é o *step* que controla a geração das *configurações restritas* que contém a regra '`step`', e dos operadores '`{_,_}`' e '`[_,_]`', veja a seção 5.5. O *sort* '`Database`' é o banco de dados de módulos gerenciados por Full Maude.

5.3 Tipos de dados

Vamos descrever agora os detalhes do processo de compilação, começando pela descrição de como tipos de dados são processados. Sabe-se que existe uma correspondência entre lógica equacional multi-sortida e gramáticas livres de contexto [50] e este fato é explorado na conversão dos tipos de dados de MSDF para assinaturas Maude. Contudo, adiamos a discussão de como tipos derivados e parametrizados para a seção 5.3.5.

Informalmente, as declarações de tipos de dados em D são usadas para gerar a assinatura Σ, convertendo cada declaração de conjunto num *sort* em S, cada declaração de inclusão de conjuntos numa inclusão de *subsorts* e cada função num operador em O.

5.3.1 Compilação de declarações de tipos

O conteúdo do módulo que define a configuração restrita ('`RCONF`', seção 2.4) é recriado como uma assinatura multi-sortida a partir de D, e para que possamos entender por que isto é necessário, vamos nos relembrar da seção 2.4 que trata da definição de configurações MRS e configurações restritas: MRS tem três construtores de configurações: '`<_,_>`', '`{_,_}` e '`[_,_]`', com uma única regra '`step`'; os operadores a regra atuam sobre o texto do programa, o que requer que *todo sort* que deve aparecer na primeira projeção da configuração deve ser um *subsort* de '`Program`'. Isto simplifica o número de elementos num módulo de sistema de Maude, também tem o efeito indesejado de colcoar todos os *sorts*, até mesmos aqueles originalmente em componentes conexos disjuntos, sob o mesmo componente conexo, o que cria restrições de regularidade e sobrecarga de operadores—uma fonte de frutração. Para evitar alguns destes problemas, o processo de compilação cria estes três operadores para cada *sort* introduzido. Este mesmo argumento é aplicado ao *sort* '`Component`', o construtor de campos '`_:_`' e a regra '`step`'. Por causa disto, uma boa parde da compilação envolve a *reconstrução* de alguns dos elementos presentes nestes módulos que agora não existem mais devido à remoção dos *sorts* '`Program`' e '`Component`'.

Formalmente, considere-se a definição de tipos de dados $\mathsf{D} = (\mathsf{S_D}, \mathsf{F_D})$ em que $\mathsf{S_D}$ é o conjunto parcialmente ordenado de conjuntos e $\mathsf{F_D}$ é o conjunto de pares de assinaturas de funções e conjunto de atributos. Para cada novo conjunto $\mathsf{s} \in \mathsf{S_D}$, um *sort* $\mathsf{s} \in \mathsf{S}$, o conjunto parcialmente ordenado de *sorts* numa teoria de reescrita Σ, é criado, com o mesmo nome. Cada inclusão de conjuntos define a ordem em S: para cada $\mathsf{s} \subseteq \mathsf{s}'$, com $\mathsf{s}, \mathsf{s}' \in \mathsf{S_D}$, os seus *sorts* correspondentes são relacionados da seguinte maneira $\mathsf{s} < \mathsf{s}'$ em Σ.

Esta funcionalidade é implementada pelas funções '`get-new-sorts`' e '`get-subsorts`', definidas no módulo '`BNF-TOOLS`', com a seguinte sintaxe:

$$\texttt{get-new-sorts} \;:\; \texttt{Set<BNF>} \rightarrow \texttt{Set<ESort>}$$
$$\texttt{get-subsorts} \;:\; \texttt{Set<BNF>} \rightarrow \texttt{Set<ESubsortDecl>}$$

em que '`Set<BNF>`' é a metarrepresentação das declarações dos tipos de dados de um módulo MSDF, '`Set<ESort>`' é a metarrepresentação das declarações de *sorts* e '`Set<ESubsortDecl>`' é a metarrepresentação das declarações de *subsort*.

Dado um conjunto parcialmente ordenado $(\mathsf{S}, \subseteq)$ de declarações de *sorts*, seja $\mathsf{S_{max}}$ o conjunto dos *supersorts maximais* em S. Ou seja, $\mathsf{S_{max}}$ contém apenas os elementos de topo de cada componente conexo de S. Para cada *sort* $\mathsf{s} \in \mathsf{S_{max}}$, o operador $\texttt{<\_,\_>} : \mathsf{s} \times \texttt{Record} \rightarrow \texttt{Conf}$ é declarado em Σ. $\mathsf{S_{max}}$ é usado ao invés de S para eivar a criação de regras e operadores desnecessários, já que todos os operadores que se aplicam a um *sort* também se aplicam aos seus subsorts.

Se o *flag step* estiver ativado, os seguintes operadores adicionais serão criados:

$$\texttt{\{\_,\_\}} \;:\; \mathsf{s} \times \texttt{Record} \rightharpoonup \texttt{Conf}$$
$$\texttt{[\_,\_]} \;:\; \mathsf{s} \times \texttt{Record} \rightharpoonup \texttt{Conf}$$

Esta funcionalidade é implementada pela função '`make-confs`' definida no módulo '`AUX-CONF-OPS`', com a seguinte assinatura:

$$\texttt{make-confs} : \texttt{Set<BNF>} \times \texttt{CFlags} \rightarrow \texttt{Set<EOpDecl>}$$

em que '`Set<EOpDecl>`' é a metarrepresentação de operadores.

5.3.2 Compilação de árvores sintáticas tipadas

Uma árvore sintática tipada usada nas transições é representada por um par '$\mathtt{t} : : : \mathtt{q}$', em que $\mathtt{t}$ é o termo e $\mathtt{q}$ é um *qid* que representa o *sort* (tipo). Este *qid* é simplesmente o nome do sort prefixado com aspas simples, tal como '`'Exp`'. Para cada *sort* $\mathtt{s} \in \mathsf{S}_{max}$, o seguinte operador é declarado em Σ:

$$\_ : : : \_ : \mathtt{s} \times \mathtt{Qid} \rightarrow \mathtt{s}$$

Esta funcionalidade é implementada pela função '`make-tst-ops`' definida no módulo '`AUX-TYPED-SYNTACTIC-TREE-OPS`', com a seguinte assinatura:

$$\texttt{make-tst-ops} \ : \ \texttt{Set<BNF>} \rightarrow \texttt{Set<EOpDecl>}$$

5.3.3 Compilação de funções

Para cada declaração de função $\mathtt{f} : \mathtt{s}_1 \times \cdots \times \mathtt{s}_n \rightarrow \mathtt{s}$ em F_D o seguinte operador é adicionado a O:

$$\mathtt{f} : \mathtt{s}_1 \times \cdots \times \mathtt{s}_n \rightarrow \mathtt{s}$$

Todos os atributos em $\mathtt{f}$ são movidos inalterados para o novo operador. A função $\mathtt{f}$ pode estar no formato *mixfix*. Neste caso o nome do operador é construído mantendo-se todos os identificadores em caixa baixa de $\mathtt{f}$ e substituindo-se os identificadores com caixa alta pelo carácter '`_`' e fazendo o domínio do operador ser o conjunto de todos os identificadores em caixa alta de $\mathtt{f}$.

A conversão de funções para operadores é feita pela função '`bnf->ops`', definida no módulo '`BNF-TOOLS`', com a seguinte assinatura:

$$\texttt{bnf->ops} : \texttt{Set<BNF>} \rightarrow \texttt{Set<EOpDecl>}$$

Em seguida, as regras '`step`' de MRS devem ser criadas. Para cada *sort* $\mathtt{s} \in \mathsf{S}_{max}$, a seguinte regra é gerada se o *flag* de *step* estiver ativado.

$$\begin{array}{rl} \textbf{crl} & \texttt{<X : : : qid(s), R>} \rightarrow \texttt{<X' : : : qid(s), R'>} \\ \textbf{if} & \{\mathtt{X} : : : \mathtt{qid}(\mathtt{s}), \mathtt{R}\} \rightarrow [\mathtt{X}' : : : \mathtt{qid}(\mathtt{s}), \mathtt{R}'] \quad [\mathtt{step}] \end{array}$$

em que X, X′ são variáveis de *sort* s e R, R′ são variáveis de *sort* '`Record`'. A função `qid(s)` converte o nome de um *sort* para um *qid*. Se o *flag* de *step* estiver desativado, nenhuma regra é gerada. As reescritas de um passo nas condições devem ser controladas por alguma outra maneira, por exemplo, através de uma estratégia de reescrita.

5.3.4 Compilação de inclusões de módulos

Vamos descrever agora o tratamento da inclusão de módulos. O módulos '`QID`' e '`MSOS-RUNTIME`' são sempre incluídos na teoria de reescrita que está sendo gerada. O primeiro é necessário devido ao uso de *qids* em árvores sintáticas tipadas em Maude através do operador '`_:::_`'; o segundo contém o suporte básico de execução para módulos MSDF, tais como as definições do *sort* '`Record`', dos *sorts* dos índices etc.

Para cada módulo m incluído com a sintaxe '`see m`', um '`including m`' correspondente é gerado. Para cada declaração de conjunto parametrizado uma inclusão da expressão de módulo que instancia o módulo parametrizado relevante é incluída.

Os detalhes desta parte são descritos na seção 5.3.5 e na tabela 5.1.

Para as inclusões implícitas descritas na seção 4.2.1, precisamos de algumas definições primeiro: seja $\mathtt{sets}(\mathsf{D})$ uma função de projeção que retorna todos os conjuntos mencionados das declarações de tipos de dados D de um módulo MSDF, $\mathtt{newsets}(\mathsf{D})$ uma função de projeção que retorna todos os *novos* conjuntos declarados em D e $\mathtt{modules}(\{\mathtt{s}_0, \ldots, \mathtt{s}_n\})$ uma função que retorne os módulos que declaram cada conjunto $\mathtt{s}_i \in \{\mathtt{s}_0, \ldots, \mathtt{s}_n\}$. Então a lista dos módulos implícitos a serem incluídos é $\mathtt{modules}(\mathtt{sets}(\mathsf{D}) \setminus \mathtt{newsets}(\mathsf{D}))$, que é a lista dos módulos que declaram os conjuntos que não são novos.

Esta funcionalidade é implementada pela função '`make-includes`', definida no módulo '`MSOS-INCLUDE-GENERATION`', com a seguinte assinatura:

$$\texttt{make-includes} : \texttt{MSOSModule} \times \texttt{CFlags} \times \texttt{Database} \rightarrow \texttt{List<EImport>}$$

em que '`List<EImport>`' é a metarrepresentação da lista de importações num módulo Full Maude. Esta função, por sua vez, depende das seguintes funções: a função '`see->import`' converte inclusões MSDF em inclusões Maude, com a assinatura.

$$\texttt{see->import} : \texttt{List<See>} \times \texttt{Database} \rightarrow \texttt{List<EImport>}$$

em que '`List<See>`' é a metarrepresentação da lista de importações em módulos MSDF.

A função ‘`generate-subsort-includes`’ lida com as importações implícitas:

$$\texttt{generate-subsort-includes} : \texttt{List<See>} \times \texttt{Set<BNF>} \times \texttt{LabelType} \times \texttt{Database} \rightarrow \texttt{List<EImport>}$$

em que ‘`LabelType`’ é a metarrepresentação de declarações de rótulos.

Finalmente, ‘`generate-parameterized-includes`’ lida com a inclusão de módulos parametrizados:

$$\texttt{generate-parameterized-includes} : \texttt{Set<BNF>} \times \texttt{Database} \rightarrow \texttt{List<EImport>}$$

Estas funções são declaradas no módulo ‘`MSOS-INCLUDE-GENERATION`’.

A função que implementa a busca de um módulo que tenha declarado um conjunto é ‘`find-declaring-module`’, definida no módulo ‘`SORT-SEARCH-TOOLS`’, com a seguinte assinatura:

$$\texttt{find-declaring-module} : \texttt{Set<ESort>} \times \texttt{Database} \rightarrow \texttt{Set<ModName>}$$

O *sort* ‘`Set<ModName>`’ é um conjunto de nomes de módulos. A função ‘`find-declaring-module`’ recebe um conjunto de *sorts* e produz o conjunto de nomes de módulos em que eles são declarados. Atualmente a ferramenta assume que existe um único módulo que defina cada *sort*. Esta função trabalha buscando todos os módulos no banco de dados gerenciado por Full Maude, extraindo a lista de *sorts* que são definidos em cada módulo e verificando se um dos *sorts* informados consta nesta lista. (Ela evita verificar os módulos abstratos $\mathsf{A}(\mathtt{s})$ internamente gerados, descritos na seção 5.3.5 que são usados para lidar com o problema de *view forwarding*.)

Para exemplificar a compilação até o momento, considere-se o seguinte fragmento MSDF que assume que o conjunto ‘`Value`’ está definido no módulo ‘`VALUE`’. Este fragmento define dois novos conjuntos, a sintaxe abstrata de duas construções, um comando *loop*, um comando para a execução paralela, duas inclusões de conjuntos e uma inclusão explícita de outro módulo MSDF.

Type	*Module*	*Sorts*
Sequences	`SEQUENCE(X :: TRIV)`	`Seq(X)`, `NeSeq(X)`
Sets	`SET(X :: TRIV)`	`Set(X)`
Lists	`LIST(X :: TRIV)`	`List(X)`
Maps	`MAP(X :: TRIV | Y :: TRIV)`	`Map(X | Y)`

Tabela 5.1: Relação entre tipos parametrizados e Full Maude

```
see SEQ .

Id . Exp .
Exp ::= Id
      | Value
      | while Exp do Exp
      | parallel (Exp, Exp) .
```

Com o *flag* de *step* ativado, ele é compilado no seguinte fragmento Maude.

```
include VALUE .          op <_,_> : Exp Record -> Conf .
include SEQ .            op {_,_} : Exp Record ~> Conf .
                         op [_,_] : Exp Record ~> Conf .
sort Id .                op _:::_ : Exp Qid -> Exp .
sort Exp .
subsort Id < Exp .       op while_do_ : Exp Exp -> Exp .
subsort Value < Exp .    op parallel : Exp Exp -> Exp .
```

5.3.5 Tipos derivados e parametrizados

Vamos estender o processo até o momento para tipos derivados e parametrizados. Posto que a compilação envolve um grande número de passos, os simples serão apresentados primeiramente, e gradualmente exibiremos os mais passos mais complexos, justificando o aumento na complexidade.

Conjuntos parametrizados em MSDF—listas, conjuntos e mapas—são convertidos em *sorts parametrizados*, que estão apenas disponíveis em Full Maude na versão 2.1.1 de Maude. Para cada tipo parametrizado existe um módulo Full Maude parametrizado (veja a seção 2.3 para uma discussão sobre eles) que implementa funcinalidade relevante. A tabela 5.1 lista a relação entre os tipos parametrizados de MSDF e os módulos e *sorts*

parametrizados de Full Maude. Lembre-se que, para instanciarmos um módulo parametrizado com um *sort* específico, precisamos criar uma visão. Com isto, a instanciação é feita importando uma expressão de módulo que envolve o módulo parametrizado e a visão desejada (seção 2.3.4). Então, cada intanciação parametrizada num módulo MSDF, a expressão de módulo correspondente que instancia o módulo parametrizado com a visão é adicionada para a seção de inclusões do módulo de sistema que está sendo gerado. Esta instanciação disponibiliza os *sorts* parametrizados que correspondem aos tipos parametrizados de acordo com a tabela 5.1. Lembre-se da seção 4.2.2 em que conjuntos parametrizados não são usados diretamente, mas através de um *conjunto equivalente*— o lado esquerdo do não terminal $\langle \mathsf{equiv} \rangle$. Este conjunto equivalente é traduzido como um *supersort* do *sort* parametrizado. Isto é necessário para que todos os elementos da álgebra do *sort* parametrizado estejam disponíveis para a álgebra do *supersort*, incluindo seus conjuntos de carga e operações.

Para formalizar isto, considere-se uma declaração de um conjunto equivalente s com o seu conjunto parametrizado $\mathsf{P}(\mathsf{s}_0, \ldots, \mathsf{s}_n)$ como $\mathsf{s} = \mathsf{P}(\mathsf{s}_0, \ldots, \mathsf{s}_n)$ em um módulo MSDF $\mathcal{M}$. Esta declaração dá origem aos seguintes componentes na teoria gerada: uma declaração de *sort* para s em Σ; uma inclusão para a instanciação do módulo parametrizado, o que depende de P de acordo com a tabela 5.1, da mesma maneira que a visão depende de $\mathsf{s}_0, \ldots, \mathsf{s}_n$; finalmente uma ordenação de *subsorts* que relaciona s e $\mathsf{P}(\mathsf{s}')$ como $\mathsf{P}(\mathsf{s}') < \mathsf{s}$ é adicionada a S.

Por exemplo, a declaração '`Channels = (Channel)Set`' gera o seguinte fragmento Maude, em que a expressão de módulo '`SET(Channel)`' é a instanciação do módulo parametrizado '`CHANNEL`' com a visão '`Channel`'. O conjunto equivalente '`Channels`' é convertido no *sort* '`Channels`' e é feito *supersort* do *sort* parametrizado '`Channel(Set)`'.

```
mod CHANNEL is
 including SET(Channel) .

 sort Channels .
 subsort Channel(Set) < Channels .
endm
```

5.3.5.1 O problema de view forwarding

A flexibilidade de MSDF gera um problema com este processo de compilação que se manifesta quando um tipo parametrizado é instanciado no mesmo módulo em que o seu conjunto de parâmetro é declarado, tal como:

```
msos CHANNEL is
 Id .
 Env = (Id, Int)Map .
sosm
```

O problema é que uma visão deve ser definida *antes* de uma instanciação de módulos—lembre-se que uma visão define um mapeamento de uma teoria (normalmente a '`TRIV`') a um módulo. Neste caso, a visão não pode ser definida porque o *sort* a que ela se refere está sendo definido no *mesmo* módulo em que a visão é necessária.

Isto é resolvido adotando-se o seguinte esquema: para cada *sort* $\mathtt{s} \in \mathsf{S}$, dois *módulos internos* são gerados e inseridos no banco de dados de Full Maude antes da compilação começar: um módulo funcional interno $\mathsf{A}(\mathtt{s})$ (de "módulo abstrato"), que contém apenas a declaração do *sort* `s` e pode ser visto como um módulo abstrato; uma visão `s`, que mapeia a teoria '`TRIV`' ao módulo $\mathsf{A}(\mathtt{s})$; a visão mapeia o *sort* '`Elt`' para o *sort* `s`.

A função que cria os módulos e visões abstratas está definida no módulo '`MSOS-SOLVER`' e tem a seguinte assinatura:

$$\texttt{insert-generated-modules\&views} : \texttt{Set<ESort>} \times \texttt{Database} \rightarrow \texttt{Database}$$

5.3.5.2 Conjuntos derivados

O tratamento de conjuntos derivados no MMT é similar ao dos conjuntos parametrizados: cada conjunto derivado é convertido num *sort* parametrizado de acordo com a tabela 5.1. A diferença no caso de conjuntos derivados é que não há nenhuma nova declaração de *sort*, nem nenhuma nova ordenação de *subsorts*. Todas as referências a '`s*`' e '`s+`' nas transições, declarações de rótulos, e definições de tipos de dados são automaticamente convertidas para '`Set(s)`' e '`NeSeq(s)`', respectivamente, durante a fase de compilação. Isto é simples, já que os tipos derivados estão sempre definidos para qualquer *sort*.

Formalmente, para cada *sort* $\mathtt{s} \in \mathsf{S}$ na assinatura do módulo de reescrita gerado, a instanciação do módulo parametrizado através da expressão '`SEQUENCE(s)`' é automatica-

mente adicionada ao módulo declarante. Esta expressão define '`Seq(s)`' e '`NeSeq(s)`' que correspondem, respectivamente, a '`s*`' e '`s+`'. Para manter a consistência de seqüências ao longo de um reticulado de *sorts*, cada inclusão de conjuntos $\mathtt{s} \subseteq \mathtt{s}'$ deve gerar não somente a relação de *subsort* $\mathtt{s} < \mathtt{s}'$, mas também a declaração de *subsort* dos seus tipos derivados: $\mathtt{Seq(s)} < \mathtt{Seq(s')}$, e $\mathtt{NeSeq(s)} < \mathtt{NeSeq(s')}$. A razão para isto é que qualquer seqüência de elementos de `s` é também uma seqüência de elementos de $\mathtt{s}'$—considere-se $(5, 4, 3, 2, 1)$, uma seqüência de números completos, que também é uma seqüência de reais, inteiros e naturais. Como a relação $\mathtt{NeSeq(s)} < \mathtt{Seq(s)}$ já é predefinida no módulo '`SEQUENCE`', não há a necessidade de relacionar `NeSeq(s)` e $\mathtt{Seq(s')}$, dado que isto já é verdade pela transtividade da relação de *subsorting*.

Para exemplificar o uso de módulos abstratos e visões, considere-se o seguinte fragmento. Ele define um conjunto '`Cmd`', digamos, de comandos e cria uma função '`seq`' que tem como único parâmetro o conjunto de seqüências não vazias de comandos.

```
msos SEQ-CMD is
 Cmd .
 Cmd ::= seq Cmd+ .
sosm
```

O seguinte código Maude é gerado (incluindo os módulos automaticamente gerados). Primeiro, o módulo abstrato '`@@ABSTRACT-Cmd`' é criado contendo apenas a declaração do *sort* '`Cmd`'.

```
fmod @@ABSTRACT-Cmd is
 sort Cmd .
endfm
```

Em seguida, a visão '`Cmd`' é gerada, que mapeia '`Elt`', definido em '`TRIV`', para '`Cmd`', definido em '`@@ABSTRACT-Cmd`'.

```
view Cmd from TRIV to @@ABSTRACT-Cmd is
 sort Elt to Cmd .
endv
```

Então, o módulo '`SEQ-CMD`' é finalmente introduzido, incluindo a expressão '`SEQUENCE(Cmd)`', que instancia o módulo parametrizado '`SEQUENCE`' com a visão '`Cmd`'.

Esta instanciação cria os *sorts* '`Seq(Cmd)`' e '`NeSeq(Cmd)`'. A função '`seq`' é convertida no operador '`seq`' e, como descrito, o conjunto '`Cmd+`' é convertido diretamente no *sort* '`NeSeq(Cmd)`'.

```
mod SEQ-CMD is
 including SEQUENCE(Cmd) .
 sort Cmd .
 op seq_ : NeSeq(Cmd) -> Cmd .
endm
```

Para exemplificar o *subsorting* de tipos derivados, considere-se o seguinte módulo que define dois conjuntos, '`Exp`' e '`Id`', e faz '`Id`' um subconjunto de '`Exp`'.

```
msos EXP is
 Id .
 Exp .
 Exp ::= Id .
sosm
```

O seguinte código gerado contém a instanciação dos módulos '`SEQUENCE(Id)`' e '`SEQUENCE(Exp)`', uma relação de *subsorting* entre '`Id`' e '`Exp`', bem como a relação de *subsorting* entre os tipos derivados.

```
mod EXP is
 including SEQUENCE(Id) .
 including SEQUENCE(Exp) .

 sort Id .
 sort Exp .
 subsort Id < Exp .
 subsort NeSeq(Id) < NeSeq(Exp) .
 subsort Seq(Id) < Seq(Exp) .
endm
```

O nível de aninhamento para tipos parametrizados é arbitrário. Por exemplo, se precisamos criar um mapa '`Ref`' que mapeia locações ('`Loc`') para ambientes ('`Env`'), que também são mapas, escrevemos:

```
Loc .
Env = (Id, Value) Map .
Ref = (Loc, Env) Map .
```

Para lidar com problemas de pré-regularidade de não-associatividade de operadores, no momento não temos tipos derivados a partir de tipos parametrizados. O capítulo 7 discute isso.

5.4 Processando declarações de rótulos

A declaração $\langle$ label $\rangle$ é usada para criar os vários operadores '`Field`' que são usados em configurações MRS. Isto é necessário também devido à remoção do *sort* '`Component`' originalmente usado em configurações MRS devido aos mesmos problemas de pré-regularidade que culminaram na remoção do *sort* '`Program`'. Além disto, uma série de *subsorts* de '`Field`' e '`Index`' também são definidos, pois são necessários na compilação das regras de transição. Para campos *read-only*, o *sort* '`ROField`' é usado; para campos de *read-write*, o *sort* '`RWField`' é usado; e para campos *write-only*, o *sort* '`WOField`' é usado. Os seguintes *sorts* para índices são definidos: para índices *read-only*, o *sort* '`RO-Index`' é usado; para índices *read-write*, ambos os *sorts* '`Pre-RW-Index`' e '`Post-RW-Index`' são definidos, relacionados às versões não-primalizadas e primalizadas dos índices, respectivamente; e finalmente, para índices *write-only*, o *sort* '`WO-Index`' é usado.

Formalmente, a compilação é feita da seguinte forma: considere-se uma declaração de rótulo L como um conjunto de declarações de campos $\{\mathsf{f}_0, \ldots, \mathsf{f}_n\}$, em que cada campo f_j é um par (i, s), com índice i e tipo de componente s. Considere-se uma função $\mathsf{sets}(\mathsf{L})$ que produz todas as segundas projeções dos campos em L e $\mathsf{indices}(\mathsf{L})$ que produz todas as primeira projeções dos campos em L. Então, para cada conjunto $\mathsf{s} \in \mathsf{sets}(\mathsf{L})$, o seu módulo correspondente $\mathsf{modules}(\mathsf{s})$ é incluído no módulo que está sendo gerado, e o seguinte operador é criado em O:

$$\_=\_ : \mathtt{Index} \times \mathsf{s} \rightharpoonup \mathtt{Field}$$

Para cada índice $\mathsf{i} \in \mathsf{indices}(\mathsf{L})$ o seguinte operador é criado em O:

$$\mathsf{i} : \rightarrow \mathsf{indexsort}(\mathsf{i})$$

em que $\mathsf{indexsort}(\mathsf{i})$ é `RO-Index` se i for um índice *read-only*, `Pre-RW-Index` se i é o índice

não-primalizado de um índice *read-write*, `Post-RW-Index` se `i` é o índice primalizado de um índice de *read-write* e `WO-Index` se `i` é um índice *write-only*.

Este procedimento é implementado pelo operador '`make-op-indices`', definido no módulo '`AUX-INDICES-OPS`'. Ele contém a seguinte assinatura:

$$\texttt{make-op-indices} : \texttt{LabelType} \rightarrow \texttt{Set<EOpDecl>}$$

Precisamos reconstruir os axiomas de pertinência da teoria MRS original que indica que o termo $\mathtt{i} = \mathtt{s}$ é um campo. Para cada campo $(\mathtt{i}, \mathtt{s}) \in \mathtt{L}$ o seguinte axioma de pertinência é criado:

$$\textbf{mb} \quad (\mathtt{i} = \mathtt{X}) : \mathtt{fieldsort}(\mathtt{i})$$

em que $\mathtt{X}$ é uma variável de *sort* $\mathtt{s}$, e $\mathtt{fieldsort}(\mathtt{i})$ é `ROField` se `i` é um índice *read-only*, `RWField` se `i` é um índice de *read-write*, e `WOField` se `i` é um índice de *write-only*. Esta funcionalidade é implementada pelo operador '`make-memberships`', definido no módulo '`MSOS-MEMBERSHIP-GENERATION`', com a seguinte assinatura:

$$\texttt{make-memberships} : \texttt{LabelType} \rightarrow \texttt{Set<EMembAx>}$$

em que '`Set<EMembAx>`' é a metarrepresentação de um conjunto de axiomas de pertinência em Full Maude.

Lembre-se que equações para o operador '`duplicate`' são usadas nas equações condicionais de pertinência que asseguram que um registro não contenha campos duplicados.

$$\texttt{duplicated} : \texttt{PreRecord} \rightharpoonup \texttt{Bool}$$

Como não existe mais um *sort* '`Component`', devemos gerar estas equações para cada *sort* que apareça numa declaração de rótulo. Assim, para cada *sort* $\mathtt{s} \in \mathtt{sets}(\mathtt{L})$, a seguinte equação é gerada:

$$\textbf{eq} \quad \mathtt{duplicated}((\mathtt{i} = \mathtt{X}), (\mathtt{i} = \mathtt{X}'), \mathtt{PR}) = \mathtt{true}.$$

em que $\mathtt{X}, \mathtt{X}'$ são variáveis de *sort* $\mathtt{s}$, $\mathtt{i}$ é uma variável de *sort* `Index`, e $\mathtt{PR}$ é uma variável de *sort* `PreRecord`. Esta funcionalidade é implementada pelo operador '`make-dup-function`', definido no módulo '`AUX-DUP-FUNCTION-EQS`', que tem a seguinte assinatura:

`make-dup-function` : `LabelType` → `Set<EEquation>`

em que '`Set<EEquation>`' é a metarrepresentação de um conjunto de equações em Full Maude:

Assim, por exemplo, uma declaração de rótulo como:

```
Label = { env : Env, st : Store, st' : Store,
          out' : Value*, ... } .
```

gera o seguinte fragmento Maude.

```
op _=_ : [Index] [Env] -> [Field] .
op _=_ : [Index] [Store] -> [Field] .
op _=_ : [Index] [Seq(Value)] -> [Field] .

op env : -> RO-Index .
op out' : -> WO-Index .
op st : -> Pre-RW-Index .
op st' : -> Post-RW-Index .

mb env = V:Env : ROField .
mb out' = V:Seq(Value) : WOField .
mb st = V:Store : RWField .
mb st' = V:Store : RWField .

eq duplicated(I:Index = X:Env,I:Index = X':Env,
              PR:PreRecord) = true .
eq duplicated(I:Index = X:Seq(Value),
              I:Index = X':Seq(Value), PR:PreRecord) = true .
eq duplicated(I:Index = X:Store,I:Index = X':Store,
              PR:PreRecord) = true .
```

5.5 Processando transições MSOS

Esta seção apresenta a implementação da transformação descrita na seção 2.4.1. Começamos descrevendo a transformação mais simples primeiro, a de transições incondicionais. Essencialmente, estamos convertendo de uma relação entre três elementos–as duas árvores sintáticas e o rótulo MSOS—para uma relação entre configurações MRS, que são tuplas contendo a ávore sintática (ou, usando a terminologia algébria, o *termo*) e o registro MRS.

A conversão das árvores sintáticas é simples: o lado esquerdo de transições MSDF via a primeira projeção da configuração do lado esquerdo e o mesmo se aplica ao lado direito. Lembre-se que, em MSDF, as árvores sintáticas contêm um *tipo* associado; esta árvore sintática tipada é convertida para tuplas de árvores sintáticas construídas pelo operador '`_:::_`', como descrito na seção 5.3. O uso de configurações restritas *versus* configurações —'`{_,_}`' e '`[_,_]`' *versus* '`<_,_>`'—é controlado pelo *flag* de *step*.

Para transformar expressões de rótulos MSOS, precisamos decompô-los em dois registros MRS, um para cada configuração MRS: o primeiro representa os campos presentes no rótulo MSOS no *início* da transição, enquanto que o segundo representa os campos presentes ao *final* da transição. A decomposição é como se segue: campos *read-only* mantêm-se inalterados, então eles aparecem tanto no início como no final; campos *read-write* são separados nas duas partes não-primalizadas e primalizadas: a parte não-primalizada é movida para o início e a primalizada para o final; finalmente, componentes de *write-only* são mais complicados, já que modelam uma "informação produzida" e não existe informação disponível no início da transição. São modelados através da *adição* do valor produzido à seqüência que corresponde a este componente. Assim, ao final da computação, este componente terá a seqüência de todos os valores produzidos.

Formalmente, considere-se uma transição como uma tupla $(\mathtt{c}, \mathtt{t}, \alpha, \mathtt{t}')$, em que $\mathtt{c}$ é a condição, $\mathtt{t}$ o lado esquerdo e, α o rótulo MSOS, e $\mathtt{t}'$ o termo resultante. No caso de transições incondicionais, $\mathtt{c}$ é simplesmente uma tautologia. A partir delas, a seguinte regra incondicional MRS é gerada; se o *flag* de *step* estiver ativado:

$$\textbf{rl} \quad \{\hat{\mathtt{t}} ::: \mathtt{qid}(\mathtt{s}), \mathtt{pre}(\hat{\alpha})\} \rightarrow [\hat{\mathtt{t}'} ::: \mathtt{qid}(\mathtt{s}), \mathtt{post}(\hat{\alpha})]$$

Se estiver desativado, configurações MRS normais são usadas, ao invés das restritas acima:

$$\textbf{rl} \quad <\hat{\mathtt{t}} ::: \mathtt{qid}(\mathtt{s}), \mathtt{pre}(\hat{\alpha})> \rightarrow <\hat{\mathtt{t}'} ::: \mathtt{qid}(\mathtt{s}), \mathtt{post}(\hat{\alpha})>$$

Aqui $\hat{\mathtt{t}}$, $\hat{\mathtt{t}'}$, e $\hat{\alpha}$ são os mesmos termos do que $\mathtt{t}$, $\mathtt{t}'$, e α, exceto que todas as metavariáveis implícitas são definidas explicitamente, de acordo com as regras definidas na seção 4.2.4. A expansão de metavariáveis implícitas é uma manipulação pruamente sintática de termos e é feita antes de qualquer processamento nas transições. Dado um termo $\mathtt{t}$, removemos todos os caracteres que não são nem letras ou algum dos símbolos '`*`' e '`+`' para criar o *sort* daquele termo. Isto é feito pelas funções '`create-transition-variables`', '`create-label-variables`', e '`create-condition-variables`', que convertem, respectivamente, termos na transição, rótulos e condições em transições MSDF.

Estas funções são implementadas no módulo '`AUTOMATIC-VARIABLES`' e têm a seguinte assinatura:

$$\begin{aligned}
\texttt{create-transition-variables} &: \texttt{QidList} \rightarrow \texttt{QidList} \\
\texttt{create-label-variables} &: \texttt{QidList} \rightarrow \texttt{QidList} \\
\texttt{create-condition-variables} &: \texttt{QidList} \rightarrow \texttt{QidList}
\end{aligned}$$

em que '`QidList`' é uma lista de *qid*s que representa os *tokens* de entrada do '`LOOP-MODE`'.

Antes de descrever o restante da compilação, é necessário uma descrição de como rótulos MSOS e registros MRS são representados algebricamente: rótulos MSOS são definidos como os *sorts* abstratos '`Label`', que representa um rótulo, '`ILabel`', que representa um rótulo não observável, '`FieldSet`', que representa um subconjunto dos campos de um rótulo e '`IFieldSet`', que representa um subconjunto não observável de campos num rótulo. Os *sorts* MRS equivalentes são, respectivamente, '`Record`', '`PreRecord`', '`IRecord`' e '`IPreRecord`'.

As funções $\mathtt{pre}$, $\mathtt{post}$ transformam de *expressões de rótulos*—`FieldSet`, `IFieldSet`—em MSOS em *expressões de registros*—`PreRecord`, `IPreRecord`—em MRS. Seja α uma expressão de rótulo genérica $\{\mathtt{f}_0, \ldots, \mathtt{f}_n\}$. Cada $\mathtt{f}_j$ é um campo, que é representado abstratamente como pares $(\mathtt{i}, \mathtt{c})$, em que $\mathtt{i}$ é o índice e $\mathtt{c}$ o componente.[1] Seja ϵ um campo "identifdade" de forma que $\{\mathtt{f}_0, \epsilon, \mathtt{f}_1\} = \{\mathtt{f}_0, \mathtt{f}_1\}$. As regras de conversão para expressões de rótulos são:

[1] Repare que este não é o mesmo que $(\mathtt{i}, \mathtt{s})$, descrito previamente na seção 5.4, que é uma *declaração de tipos*, com $\mathtt{s}$ o *sort* do componente indexado por $\mathtt{i}$.

$$\begin{aligned} \mathtt{pre}(\{\mathsf{f}_0, \ldots, \mathsf{f}_n\}) &= \{\mathtt{pre}(\mathsf{f}_0), \ldots, \mathtt{pre}(\mathsf{f}_n)\} \\ \mathtt{post}(\{\mathsf{f}_0, \ldots, \mathsf{f}_n\}) &= \{\mathtt{post}(\mathsf{f}_0), \ldots, \mathtt{post}(\mathsf{f}_n)\} \end{aligned}$$

em que "..." é apenas uma forma abreviada de dizer que a função se aplica sobre todos os campos de um rótulo.

As variáveis que representam o "restante do rótulo" são convertidas da seguinte maneira:

$$\begin{aligned} \mathtt{pre}(\mathsf{U}) &= \tilde{\mathsf{U}} \\ \mathtt{post}(\mathsf{U}) &= \tilde{\mathsf{U}} \\ \mathtt{pre}(\mathsf{X}) &= \tilde{\mathsf{X}} \\ \mathtt{post}(\mathsf{X}) &= \tilde{\mathsf{X}'} \end{aligned}$$

em que U é uma variável do *sort* `IFieldSet` e X, X' são variáveis do *sort* `FieldSet`. O resultado da conversão gera variáveis $\tilde{\mathsf{X}}$ e $\tilde{\mathsf{X}'}$ do *sort* `PreRecord`, e $\tilde{\mathsf{U}}$, do *sort* `IPreRecord`. Como uma varíavel dos *sorts* '`Label`' ou '`ILabel`' é equivalente à expressão de rótulo $\{\mathsf{V}\}$ sendo V uma variável do *sorts* '`FieldSet`' ou '`IFieldSet`', não há necessidade de criarmos equações para $\mathtt{pre}$ e $\mathtt{post}$ que lidem sobre rótulos inteiros.

Vamos descrever agora as equações que lidam com um campo específico (i, c). Se i for um índice *read-only*:

$$\begin{aligned} \mathtt{pre}(\mathsf{i}, \mathsf{c}) &= (\mathsf{i}, \mathsf{c}) \\ \mathtt{post}(\mathsf{i}, \mathsf{c}) &= (\mathsf{i}, \mathsf{c}) \end{aligned}$$

Se i for o índice *não-primalizado* de um índice *read-write*:

$$\begin{aligned} \mathtt{pre}(\mathsf{i}, \mathsf{c}) &= (\mathsf{i}, \mathsf{c}) \\ \mathtt{post}(\mathsf{i}, \mathsf{c}) &= \epsilon \end{aligned}$$

Se i' for o índice *primalizado* de um índice *read-write*:

$$\begin{aligned} \mathtt{pre}(\mathsf{i}', \mathsf{c}) &= \epsilon \\ \mathtt{post}(\mathsf{i}', \mathsf{c}) &= (\mathsf{i}', \mathsf{c}) \end{aligned}$$

Se $\mathtt{i}'$ for um índice *write-only*:

$$\begin{aligned} \mathtt{pre}(\mathtt{i}', \mathtt{c}) &= (\mathtt{i}', \mathtt{V}) \\ \mathtt{post}(\mathtt{i}', \mathtt{c}) &= (\mathtt{i}', (\mathtt{V}, \mathtt{c})) \end{aligned}$$

em que $\mathtt{V}$ é uma nova variável do *sort* $\mathtt{s}$. Lembre-se que a única monóide livre atualmente aceita por MMT é a seqüência finita com identidade '`()`', e operação binária '`_,_`'.

Para transições condicionais, o único tipo de condição que precisa atenção é a *transição na condição*; os outros tipos são convertidos sem modificação de MSDF para Maude, já que eles estão presentes em Maude—são equações condicionais, instanciações de variáveis e predicados. Transições condicionais são convertidas em regras condicionais, em que as transições nas condições são convertidas em reescritas nas condições usando uma conversão um pouco diferente da conversão de transições incondicionais. A diferença é em relação a componentes *write-only*: no início de uma transição condicional o componente é a seqüência vazia '`()`' de forma que qualquer informação produzida está presente na conclusão da transição.

Formalmente, uma transição condicional $(\mathtt{c}, \mathtt{t}, \alpha, \mathtt{t}')$ é convertida como se segue. Iremos considerar a condição $\mathtt{c}$ uma conjunção de condições $\mathtt{c}_0, \ldots, \mathtt{c}_n$. Se o *flag* de *step* estiver ativado, a seguinte regra de reescrita conditional é gerada:

$$\begin{aligned} \textbf{crl}\ & \{\mathtt{t} ::: \mathtt{qid}(\mathtt{s}), \mathtt{pre}(\alpha)\} \to [\mathtt{t}' ::: \mathtt{qid}(\mathtt{s}), \mathtt{post}(\alpha)] \\ \textbf{if}\ & \mathtt{cond}(\mathtt{c}_0, \ldots, \mathtt{c}_n) \end{aligned}$$

Por outro lado, se o *flag* de *step* estiver desativado, a seguinte regra é gerada:

$$\begin{aligned} \textbf{crl}\ & \texttt{<}\mathtt{t} ::: \mathtt{qid}(\mathtt{s}), \mathtt{pre}(\alpha)\texttt{>} \to \texttt{<}\mathtt{t}' ::: \mathtt{qid}(\mathtt{s}), \mathtt{post}(\alpha)\texttt{>} \\ \textbf{if}\ & \mathtt{cond}(\mathtt{c}_0, \ldots, \mathtt{c}_n) \end{aligned}$$

A função $\mathtt{cond}(\mathtt{c})$ converte as condições nas transições em condições de reescrita de acordo com as seguintes regras:

$$\mathtt{cond}(\mathtt{c}_0, \ldots, \mathtt{c}_n) = \mathtt{cond}(\mathtt{c}_0) \wedge \cdots \wedge \mathtt{cond}(\mathtt{c}_n);$$

Para cada condição $\mathtt{c}_i$, a conversão é a seguinte. Se $\mathtt{c}_i$ é uma transição condicional $(\mathtt{t}, \alpha, \mathtt{t}')$, então é numa reescrita condicional. Se o *flag* de *step* estiver ativado, esta

condição é a seguinte:

$$\{\mathtt{t}:::\mathtt{qid}(\mathtt{s}), \mathtt{pre}'(\alpha)\} \rightarrow [\mathtt{t}':::\mathtt{qid}(\mathtt{s}), \mathtt{post}'(\alpha)]$$

Caso contrário, a condição é a seguinte:

$$<\mathtt{t}:::\mathtt{qid}(\mathtt{s}), \mathtt{pre}'(\alpha)> \rightarrow <\mathtt{t}':::\mathtt{qid}(\mathtt{s}), \mathtt{post}'(\alpha)>$$

em que $\mathtt{s}$ é o *sort mínimo* aplicável a $\mathtt{t}$, implicando em pré-regularidade.

Se $\mathtt{c}_\mathtt{i}$ não é uma transição condicional:

$$\mathtt{cond}(\mathtt{c}_\mathtt{i}) = \mathtt{c}_\mathtt{i}$$

As funções $\mathtt{pre}'$ e $\mathtt{post}'$ trabalham da mesma forma que $\mathtt{pre}$ e $\mathtt{post}$. Estas equações seguem as equações para as funções $\mathtt{pre}$ e $\mathtt{post}$. As mesmas observações se aplicam.

$$\begin{aligned}
\mathtt{pre}'(\{\mathtt{f}_0, \ldots, \mathtt{f}_n\}) &= \{\mathtt{pre}'(\mathtt{f}_0), \ldots, \mathtt{pre}'(\mathtt{f}_n)\} \\
\mathtt{post}'(\{\mathtt{f}_0, \ldots, \mathtt{f}_n\}) &= \{\mathtt{post}'(\mathtt{f}_0), \ldots, \mathtt{post}'(\mathtt{f}_n)\} \\
\mathtt{pre}'(\mathtt{U}) &= \tilde{\mathtt{U}} \\
\mathtt{post}'(\mathtt{U}) &= \tilde{\mathtt{U}} \\
\mathtt{pre}'(\mathtt{X}) &= \tilde{\mathtt{X}} \\
\mathtt{post}'(\mathtt{X}) &= \tilde{\mathtt{X}}'
\end{aligned}$$

Para campos, a única diferença está no tratamento de campos *write-only*. Se $\mathtt{i}$ for um índice de *write-only*:

$$\begin{aligned}
\mathtt{pre}'(\mathtt{i}, \mathtt{c}) &= (\mathtt{i}, \mathtt{c}) \\
\mathtt{post}'(\mathtt{i}, \mathtt{c}) &= (\mathtt{i}, \mathtt{c})
\end{aligned}$$

Se $\mathtt{i}$ é o índice não-primalizado de um índice de *read-write*:

$$\begin{aligned}
\mathtt{pre}'(\mathtt{i}, \mathtt{c}) &= (\mathtt{i}, \mathtt{c}) \\
\mathtt{post}'(\mathtt{i}, \mathtt{c}) &= \epsilon
\end{aligned}$$

Se $\mathtt{i}'$ é o índice primalizado de um índice *read-write*:

$$\begin{aligned} \mathtt{pre}'(\mathtt{i}', \mathtt{c}) &= \epsilon \\ \mathtt{post}'(\mathtt{i}', \mathtt{c}) &= (\mathtt{i}', \mathtt{c}) \end{aligned}$$

Se $\mathtt{i}'$ for um índice de *write-only*:

$$\begin{aligned} \mathtt{pre}'(\mathtt{i}', \mathtt{c}) &= (\mathtt{i}', ()) \\ \mathtt{post}'(\mathtt{i}', \mathtt{c}) &= (\mathtt{i}', \mathtt{c}) \end{aligned}$$

Finalmente, vamos descrever alguns aspectos práticos desta conversão. Primeiro, as transições devem ser normalizadas de acordo com o seguinte: todas as relações `-->` são convertidas em `-{U}->`, a metavariável '`-`' em expressões de rótulo é convertida em '`U`', e a metavariável '`...`' para '`X`'.

A função principal na conversão de transições MSDF para regras de reescrita é '`make-rewriting-rules`' no módulo '`MSOS-RULE-GENERATION`', com a seguinte assinatura:

$$\texttt{make-rewriting-rules} : \texttt{Set<Transition>} \times \texttt{LabelType} \times \texttt{CFlags} \rightarrow \texttt{RuleSet}$$

em que '`Set<Transition>`' é a metarrepresentação de transições MSDF, '`LabelType`', a metarrepresentação da declaração de rótulo presente no mesmo módulo que contém o conjunto de transições e '`CFlags`' são os *flags* de compilação. A imagem é '`RuleSet`', que é a metarrepresentação de um conjunto de regras de reescrita. Esta função itera sobre as transições em '`Set<Transition>`'. Para que possamos gerar as configurações iniciais e finais em cada regra, esta função chama as funções '`lhs-conf`' e '`rhs-conf`', definidas no módulo '`AUX-CONF-UTILS`', com a seguinte assinatura:

$$\texttt{lhs-conf} : \texttt{Term} \times \texttt{QidTerm} \times \texttt{ConfLocation} \times \texttt{CFlags} \rightarrow \texttt{Term}$$

em que '`rhs-conf`' tem a mesma assinatura. Aqui, seguindo a ordem dos parâmetros, '`Term`' é a árvore sintática no início da transição, '`Qid`' é o tipo desta árvore sintática, '`Term`' é a árvore sintática resultante, '`ConfLocation`' indica se esta configuração está sendo gerada numa conclusão ou numa condição, e finalmente, '`CFlags`' são os *flags* de compilação. O *sort* de imagem é '`Term`', que é a metarrepresentação de uma confição MRS.

Finalmente, a conversão de condições é feita pela função '`convert-condition`' no

módulo '`MSOS-RULE-GENERATION`', com a seguinte assinatura:

$$\texttt{convert-condition} : \texttt{MSOS-Condition} \times \texttt{IsComp?} \times \texttt{CFlags} \rightarrow \texttt{Condition}$$

em que o parâmetro '`MSOS-Condition`' é a metarrepresentação da conjunção de condições de uma transição específica, '`IsComp?`' é um *flag* que indica se a transição em que estas condições ocorrem usa composição de rótulos para que possa ser tratada de acordo, '`CFlags`' são os *flags* de compilação e o *sort* de imagem é '`Condition`', a metarrepresentação das condições em regras de reescrita.

As variáveis dependentes de fonte em MMT não são checadas em transições MSDF, mas somente após a compilação terminar—nas regras geradas. Esta opção foi feita para simplificar o processo de verificação, já que o processo de compilação pode introduzir novas variáveis nas regras de reescritas e é mais simples verificar pela dependência de fonte em transições não-rotuladas, como é o caso de regras de reescrita (os rótulos que existem são apenas decorativos e não contém metavariáveis).

O processo de verificação segue a definição de variáveis dependentes da fonte: todas as variáveis na fonte da conclusão são dependentes de fonte; se todas as variáveis na fonte de uma condição são dependentes de fonte, então todas as variáveis na conclusão são dependentes de fonte. Formalmente, considere uma regra de reescrita $\mathtt{r}$, como a abaixo:

$$\mathtt{t} \rightarrow \mathtt{t}' \Leftarrow \mathtt{c}_1 \wedge \cdots \wedge \mathtt{c}_n$$

Seja $\mathtt{vars}(\mathtt{t})$ o conjunto de variáveis no termo $\mathtt{t}$. Seja $\mathtt{lhs}(\mathtt{t})$ o conjunto de variáveis na fonte de $\mathtt{t}$ e $\mathtt{rhs}(\mathtt{t})$ o conjunto na conclusão. Definimos incrementalmente $\mathtt{sd}(\mathtt{r})$, o conjunto de variáveis dependentes de fonte de uma regra $\mathtt{r}$ como:

- adicione $\mathtt{vars}(\mathtt{t})$ a $\mathtt{sd}(\mathtt{r})$;
- para cada condição $\mathtt{c}_i$, da esquerda para a direita, fazemos o seguinte: se $\mathtt{lhs}(\mathtt{c}_i) \subseteq \mathtt{sd}(\mathtt{r})$ então adicione $\mathtt{rhs}(\mathtt{c}_i)$ a $\mathtt{sd}(\mathtt{r})$;
- adicione $\mathtt{lhs}(\mathtt{c}_i)$ a $\mathtt{sd}(\mathtt{r})$ se $\mathtt{c}_i$ é uma instanciação de variáveis (*matching equation*).

A verificação de variáveis dependentes de fonte é feita pela função '`source-dependent`', definida no módulo '`SOURCE-DEPENDENCY-CHECK`', com a seguinte assinatura:

$$\texttt{source-dependent} : \texttt{Rule} \rightarrow \texttt{Set<Var>}$$

Capítulo 6

Estudos de caso

Este capítulo descreve diversas aplicações de MMT: a seção 6.1 descreve *Constructive MSOS* [10] e seu uso na definição formal de linguagens de programação; seção 6.2 descreve a semântica de uma linguagem de ordem normal, Mini-Freja [43]; seção 6.3 descreve diversos algoritmos distribuídos de [51] especificados em MSDF e verificados em Maude.

6.1 Constructive MSOS

A técnica *Constructive MSOS* (CMSOS) [10] é uma técnica para a especificação de linguagens de programação baseada na idéia de que cada construção de uma linguagem deve ser especificada em um módulo separado, e que uma especificação completa de uma linguagem é baseada na combinação destes módulos. Uma técnica relacionada, chamada *Incremental Action Semantics*, que usa as mesmas idéias e foi desenvolvida com o *framework* de *Action Semantics*, é descrita em [52].

Uma outra generalização de CMSOS é usar construções *neutras* em relação a uma linguagem de programação específica. Esta generalização permite um alto grau de reuso destes módulos numa vasta gama de linguagens de programação. As notas de aula de Mosses [10] contêm um conjunto proposto destas construções abstratas, que implementamos. Exemplificamos também o uso destas construções abstratas para dar a semântica formal de duas linguagens de programação diferentes: um subconjunto de ML, descrito em [10] (seção 6.1.2) e um subconjunto de Java, chamado de MiniJava, baseado na linguagem descrita em [53] (seção 6.1.3).

6.1.1 As construções CMSOS

Esta seção descreve o conjunto CMSOS de construções proposto por Mosses. Cada subseção descreve um aspecto particular da linguagem CMSOS. Seguimos [10] na ordem de apresentação que divide a semântica de CMSOS em cinco aspectos: expressões, declarações, abstrações, comandos e concorrência. Optamos por não apresentar todo CMSOS aqui, selecionando apenas aquelas construções que são necessárias para as semânticas de ML e MiniJava.

Antes de prosseguir, devemos descrever a terminologia de módulos CMSOS: lembre-se do capítulo 3 em que o MSOS Tool de Mosses contém uma biblioteca que divide cada módulo em seu próprio arquivo, de uma maneira hierárquica, usando diretórios. Por exemplo, o diretório '`Cons/`' contém todas as *construções* de CMSOS. Dentro de '`Cons/`', o diretório '`Exp/`' contém todas as construções relacionadas a *expressões*. Dentro de '`Exp/`', um dos diversos diretórios é '`tup/`', que contém o módulo que define a sintaxe de tuplas genéricas. Módulos MSDF devem ter um nome, então optamos por usar a hierarquia de diretório como o nome do módulo. Mosses também divide cada construção em três módulos separados: um para a sintaxe abstrata, outro para a semântica estática e outro para a semântica dinâmica. Para simplificar, optamos por combiná-los em um único módulo.

Esta seção descreve alguns módulos selecionados da especificação CMSOS completa, necessários para dar a semântica das outras linguagens neste capítulo, ML (seção 6.1.2) e MiniJava (seção 6.1.3). O apêndice A contém o restante das construções.

6.1.1.1 Expressões

As expressões são os blocos básicos das construções de programas CMSOS e contêm as seguintes construções: tuplas, condicionais, aplicação de operadores e aplicação de identificadores. Começamos mostrando a semântica abstrata de tuplas de expressões, inicialmente a sua forma mais geral e em seguida uma versão mais restrita.

Começamos definindo os módulos gerais que definem o conjunto de expressões, '`Exp`', e valores, '`Value`'. O módulo '`Value`' define o conjunto de valores, além de uma função '`apply-op`', que recebe uma operação '`Op`', e uma seqüência de valores, '`Value*`'. O conjunto '`Op`', definido no módulo '`Cons/Op`', contém todas as operações primitivas sobre valores. Para simplificar as coisas, evitamos definir regras para operações extremamente óbvias, tais como operações de aritmética e comparação: estas são definidas fora da

especificação, como iremos ver mais tarde. A idéia é que o especificador não deve se preocupar na definição de coisas como adição de dois inteiros e assim por diante.

```
msos Cons/Op is
 Op .
sosm

msos Value is
 Value .
 Value ::= apply-op (Op, Value*) .
sosm
```

Em seguida, definimos o conjunto de expressões, '`Exp`', no módulo '`Cons/Exp`'. Expressões são computadas em valores, daí a inclusão de conjuntos.

```
msos Cons/Exp is
 Exp .
 Exp ::= Value .
sosm
```

O módulo '`Cons/Exp/tup`' abaixo define uma forma genérica de tuplas em que cada elemento é selecionado não deterministicamente para ser computado.

```
msos Cons/Exp/tup is
 Exp ::= tup Exp* .
 Value ::= tup Value* .

                              Exp -{...}-> Exp'
 -- ------------------------------------------------------------
 (tup (Exp*,Exp,Exp*')) : Exp -{...}-> tup (Exp*,Exp',Exp*') .
sosm
```

Uma variante seqüencial tem a restrição de que elementos devem ser computados da esquerda para a direita.

```
msos Cons/Exp/tup-seq is
```

```
 Exp ::= tup-seq Exp* .
 Value ::= tup Value* .

                  Exp -{...}-> Exp'
 -- ------------------------------------------------------------
 (tup-seq (Value*, Exp, Exp*)) : Exp
                       -{...}-> (tup-seq (Value*, Exp', Exp*)) .
 (tup-seq Value*) : Exp --> tup (Value*) .
sosm
```

6.1.1.2 Declarações

As construções declarativas definem amarrações e computam expressões dentro do contexto de um conjunto de amarrações. Uma amarração associa um identificador a um valor "amarrável" (*bindable*)—um valor que pode ser parte de uma amarração. O conjunto '`Bindable`' é o conjunto de valores amarráveis.

```
msos Data/Bindable is
 Bindable .
sosm
```

Para definir os identificadores, definimos o conjunto '`Id`' no módulo abaixo:

```
msos Id is
 Id .
 Id ::= Bindable .
sosm
```

Podemos agora definir *ambientes* de amarrações, que são usados para associar identificadores com valores amarráveis. Um ambiente An environment '`Env`' é um conjunto parametrizado, um mapa de '`Id`' para '`Bindable`'.

```
msos Data/Env is
 Env .

 Env = (Id, Bindable) Map .
sosm
```

A definição de identificadores faz uso de um componente de *read-only* indexado por '`env`'. A computação de uma amarração busca seu valor no componente '`Env`'.

```
msos Cons/Id is
 Label = {env : Env, ...} .

     Bindable := lookup (Id, Env)
 -- --------------------------------
 Id : Id -{env = Env, -}-> Bindable .
sosm
```

O conjunto '`Dec`' é o conjunto de declarações (*declarations*) na linguagem. Declarações computam em amarrações, daí a inclusão de conjuntos.

```
msos Cons/Dec is
 Dec .
 Dec ::= Env .
sosm
```

A construção '`bind`' declara a amarração de um identificador '`Id`' e o valor resultante da computação de uma expressão '`Exp`'.

```
msos Cons/Dec/bind is
 Dec ::= bind Id Exp .

                 Exp -{...}-> Exp'
 -- ----------------------------------------
 (bind Id Exp) : Dec -{...}-> (bind Id Exp') .

 (bind Id Value) : Dec --> Id |-> Value .
sosm
```

6.1.1.3 Abstrações

Descrevemos aqui as construções associadas com abstrações procedurais e amarrações recursivas. Começamos definindo o conjunto de abstrações e o conjunto de seus parâmetros formais.

```
msos Cons/Abs is
 Abs .
sosm

msos Cons/Par is
 Par .
sosm
```

Quando usado para definir uma linguagem onde abstrações são expressões, o seguinte módulo deve ser usado:

```
msos Cons/Exp/Abs is
 Exp ::= Abs .
 Value ::= Abs .
sosm
```

A abstração é definida usando-se a sintaxe abstrata definida no módulo '`Cons/Abs/abs-Exp`'. O conjunto '`Passable`' é o conjunto de valores que podem ser usados como argumentos. A aplicação de uma abstração a um destes valores cria uma declaração '`local`', com o valor que está sendo amarrado ao parâmetro através da construção '`app`', descrita no módulo '`Cons/Dec/app`'.

```
msos Cons/Abs/abs-Exp is
 see Cons/Exp/local, Cons/Dec/app .
 see Cons/Exp/Abs, Cons/Exp/app .

 Abs ::= abs Par Exp .

 (app (abs Par Exp) Passable) : Exp -->
                (local (app Par Passable) Exp) .
sosm
```

Definimos abaixo o conceito de *closure*, que é, em essência, uma abstração com um ambiente:

```
msos Cons/Exp/close is
 see Cons/Abs/closure .

 Exp ::= close Abs .

 Label = {env : Env, ...} .

 (close Abs) : Exp -{env = Env,-}-> (closure Env Abs) .
sosm
```

O módulo '`Cons/Abs/closure`' contém a definição da aplicação de uma *closure* a um valor. Em essência, o ambiente da *closure* é usado como uma "declaração externa" da construção '`local`'.

```
msos Cons/Abs/closure is
 see Cons/Exp/local, Cons/Abs .
 see Cons/Exp/app, Cons/Exp/Abs .

 Abs ::= closure Dec Abs .

 (app (closure Dec Abs) Passable) : Exp -->
                    (local Dec (app Abs Passable)) .
sosm
```

6.1.1.4 Comandos

Comandos formam o aspecto imperativo de CMSOS. Começamos pela definição do conjunto de comandos, '`Cmd`'. Todos os comandos computam em um valor final, '`skip`', que é o comando inócuo.

```
msos Cons/Cmd is
 Cmd .
 Cmd ::= skip .
sosm
```

Para executar uma seqüência de comandos, usamos a construção '`seq-n`', que recebe como parâmetro uma lista não-vazia de comandos, '`Cmd+`', e computa cada comando em

ordem. À medida que cada comando termina, ou seja, é computado em '`skip`', ele é removido desta tupla.

```
msos Cons/Cmd/seq-n is
 Cmd ::= seq Cmd+ .

                     Cmd -{...}-> Cmd'
 -- ------------------------------------------------
 (seq (Cmd, Cmd*)) : Cmd -{...}-> (seq (Cmd', Cmd*)) .

 (seq (skip, Cmd+)) : Cmd --> (seq Cmd+) .

 (seq skip) : Cmd --> skip .
sosm
```

A construção '`effect`' é a ponte entre expressões e comandos. Ela computa a expressão para um valor e remove este valor. Idealmente, a expressão terá modificado algum componente *write-only* ou *read-write*, tal como a operação de alterar um valor em memória. Uma vez obtido o valor final pela computação da expressão, o comando computa em '`skip`'.

```
msos Cons/Cmd/effect is
 Cmd ::= effect Exp .

              Exp -{...}-> Exp'
 -- ------------------------------------
 (effect Exp) : Cmd -{...}-> effect Exp' .

 (effect Value) : Cmd --> skip .
sosm
```

6.1.1.5 Concorrência

Definimos nesta seção o aspecto de concorrência de especificações CMSOS. Vamos começar definindo o conceito de um programa concorrente, ou "sistema," representado pelo conjunto '`Sys`'.

```
msos Cons/Sys is
 Sys .
sosm
```

O conjunto de *threads* que roda em um sistema é composto pela função '`conc`'. Duas regras selecionam não-deterministicamente qual projeção deve ser computada a cada passo. Optamos por seguir a especificação de Mosses aqui, mas podemos também definir a função '`conc`' como sendo comutativa, usando apenas uma regra. Em qualquer caso, o efeito final é um modelo *interleaving* (alternância) de concorrência.

```
msos Cons/Sys/conc is
 Sys ::= conc Sys Sys .

                  Sys1 -{...}-> Sys1'
 -- ------------------------------------------------
 (conc Sys1 Sys2) : Sys -{...}-> (conc Sys1' Sys2) .

                  Sys2 -{...}-> Sys2'
 -- ------------------------------------------------
 (conc Sys1 Sys2) : Sys -{...}-> (conc Sys1 Sys2') .
sosm
```

6.1.2 ML

ML é um subconjunto de Concurrent ML [2] e é o primeiro exemplo de como dar a semântica de uma linguagem em termos de CMSOS. A conversão de ML para CMSOS descrita nesta seção segue a descrita em [10].

A transformação da sintaxe de ML na a sintaxe abstrata de CMSOS é feita usando uma equação, onde o equivalente CMSOS de uma construção ML `c` é descrito entre colchetes duplos (⟦c⟧). A equação especifica como cada componente de uma construção é convertido em construções CMSOS. A transformação de uma consrução `f` que contenha parâmetros `x` e `y` é escrita como: ⟦f(x,y)⟧ = m(⟦x⟧,⟦y⟧) significando que a construção `f` é converitda numa construção CMSOS '`m`' que recebe como argumentos os argumentos convertidos `x` e `y` de `f`.

Esta seção apresenta apenas um subconjunto da compilação de ML para CMSOS, e o apêndice B dá a especificação completa.

6.1.2.1 Expressões

Começamos descrevendo expressões ML a construções CMSOS equivalentes. Primeiro, precisamos reunir todos os módulos CMSOS necessários para as definições de expressões em ML. Isto é feito criando-se um módulo '`Lang/ML/Exp`'. O módulo contém referências explícitas para todas as construções CMSOS necessárias. Também define que o conjunto de valores ('`Value`') e operadores ('`Op`') são "amarráveis" em ambientes, que o conjunto de valores é "passável" para abstrações procedurais e que o conjunto de operadores contém as constantes '`plus`', '`times`', etc.

```
msos Lang/ML/Exp is
 see Cons/Prog, Cons/Prog/Exp .

 see Cons/Exp, Cons/Exp/Boolean, Cons/Exp/Int,
     Cons/Exp/Id, Cons/Exp/cond, Cons/Exp/app-Op,
     Cons/Exp/app-Id, Cons/Exp/tup, Cons/Exp/tup-seq .

 see Cons/Arg, Cons/Arg/Exp .

 see Cons/Op .

 see Cons/Id .

 Bindable ::= Value | Op .

 Op ::= plus | times | minus | eq | lt | gt .

 Passable ::= Value .
sosm
```

O seguinte módulo contém as definições básicas para expressões ML. Ele é definido como um módulo de sistema que inclui o módulo MSDF '`Lang/ML/Exp`'. Lembre-se que devemos definir a operação '`apply-op`' externamente e isto é feito aqui para cada constante '`Op`' declarada no módulo '`Lang/ML/Exp`'. Em seguida, criamos o ambiente inicial com as amarrações de identificadores para operadores. Reusamos os nomes dos operadores como identificadores, criados pela função de coerção '`ide`', no mapeamento.

```
mod Lang/ML/Exp' is
 including Lang/ML/Exp .
 including QID .

 vars i1 i2 : Int .

 eq apply-op (plus, (i1, i2)) = i1 + i2 .
 eq apply-op (minus, (i1, i2)) = i1 - i2 .
 eq apply-op (times, (i1, i2)) = i1 * i2 .
 eq apply-op (eq, (i1, i2)) = if i1 == i2 then tt else ff fi .
 eq apply-op (lt, (i1, i2)) = if i1 < i2 then tt else ff fi .
 eq apply-op (gt, (i1, i2)) = if i1 > i2 then tt else ff fi .

 op ide : Qid -> Id .
 op ide : Op -> Id .
 op op : Qid -> Op .

 eq init-env = (ide(eq) |-> eq +++ ide(lt) |-> lt +++
                ide(gt) |-> gt +++ ide(plus)  |-> plus +++
                ide(times) |-> times +++ ide(minus) |-> minus) .


 eq op ('+) = plus .   eq op ('*) = times .
 eq op ('-) = minus .  eq op ('<) = lt .
 eq op ('>) = gt .     eq op ('=) = eq .
endm
```

▶ *Expressões completas*

Estas são as regras para expressões completas em ML. Omitimos as regras para a conversão de identificadores, constantes especiais (i.e., números), aplicações de expressões e expressões infixadas.

$$\langle \texttt{exp} \rangle \rightarrow \langle \texttt{exp} \rangle \text{ `}\texttt{andalso}\text{' } \langle \texttt{exp} \rangle \mid \langle \texttt{exp} \rangle \text{ `}\texttt{orelse}\text{' } \langle \texttt{exp} \rangle$$
$$\mid \text{`}\texttt{if}\text{' } \langle \texttt{exp} \rangle \text{ `}\texttt{then}\text{' } \langle \texttt{exp} \rangle \text{ `}\texttt{else}\text{' } \langle \texttt{exp} \rangle$$

$[\![e_0$ `andalso` $e_1]\!] = $ `cond` $[\![e_0]\!]$ $[\![e_1]\!]$ `ff`

$[\![e_0$ `orelse` $e_1]\!] = $ `cond` $[\![e_0]\!]$ `tt` $[\![e_1]\!]$

$[\![$`if` e_0 `then` e_1 `else` $e_2]\!] = $ `cond` $[\![e_0]\!]$ $[\![e_1]\!]$ $[\![e_2]\!]$

6.1.2.2 Declarações

Para declarações, vamos introduzir o módulo MSDF relevante que contém todas as construções CMSOS necessárias.

```
msos Lang/ML/Dec is
 see Lang/ML/Exp' .

 see Cons/Prog, Cons/Prog/Dec .

 see Cons/Dec, Cons/Dec/bind, Cons/Dec/simult-seq,
     Cons/Dec/accum, Cons/Dec/local .

 see Cons/Exp, Cons/Exp/local .
sosm
```

► *Expressões "let"*

Começamos estendendo as expressões atômicas com a expressão '`let-in-end`', que é mapeada na construção CMSOS '`local`'.

$\langle$ `exp` $\rangle \rightarrow$ '`let`' $\langle$ `dec` $\rangle$ '`in`' $\langle$ `exp` $\rangle$ '`end`'

Seja `d` uma metavariável sobre $\langle$ `dec` $\rangle$.

$[\![$`let d in` e `end`$]\!] = $ `local` $[\![$`d`$]\!]$ $[\![e]\!]$

► *Amarrações de valores*

Em seguida, as declarações são definidas. Amarrações de valores são convertidas na construção CMSOS '`bind`'.

$\langle\, \mathsf{dec} \,\rangle \rightarrow$ 'val' $\langle\, \mathsf{vid} \,\rangle$ '=' $\langle\, \mathsf{exp} \,\rangle$

$[\![\texttt{val i = e}]\!] \;=\; \texttt{bind}\; [\![\texttt{i}]\!]\; [\![e]\!]$

6.1.2.3 Imperativos

ML não tem o conceito de um "comando," já que tudo é uma expressão nesta linguagem, mas usa aspectos imperativos. Optamos por mostrar aqui as regras de conversão para atribuição de valores e o comando de *loop*.

O módulo 'Lang/ML/Cmd' define os "comandos" na linguagem ML:

```
msos Lang/ML/Cmd is
 see Lang/ML/Dec .

 see Cons/Cmd, Cons/Cmd/seq-n, Cons/Cmd/effect, Cons/Cmd/while .

 see Cons/Exp, Cons/Exp/seq-Cmd-Exp, Cons/Exp/seq-Exp-Cmd,
     Cons/Exp/assign-seq, Cons/Exp/ref, Cons/Exp/assigned .

 see Cons/Var, Cons/Var/alloc, Cons/Var/deref .

 Storable ::= Value .
sosm
```

Em seguida adicionamos uma outra definição "externa", que é a equação que aloca uma nova célula numa memória.

```
mod Lang/ML/Cmd' is
 including Lang/ML/Cmd .

 var Store : Store .
 eq new-cell (Store) = cell (length (Store) + 1) .
endm
```

▶ *Atribuição*

Já que uma atribuição em ML não contém um valor final, usamos a construção '`seq-Cmd-Exp`' para primeiro executar a atribuição e depois para retornar a tupla vazia ('`tup()`'). A atribuição propriamente dita é feita computando-se a expressão atribuída e em seguida usando-se a construção '`assign-seq`'.

$$\langle \mathsf{exp} \rangle \rightarrow \langle \mathsf{exp} \rangle \text{ ':=' } \langle \mathsf{exp} \rangle$$

$$[\![e_0 \texttt{ := } e_1]\!] = \texttt{seq (effect (assign-seq (deref } [\![e_0]\!] \texttt{) } [\![e_1]\!] \texttt{)) tup()}$$

► *Loops*

Finalmente, adicionamos uma construção típica de uma linguagem imperativa, o comando de *loop*.

$$\langle \mathsf{exp} \rangle \rightarrow \langle \mathsf{while} \rangle \langle \mathsf{exp} \rangle \text{ 'do' } \langle \mathsf{exp} \rangle$$

$$[\![\texttt{while } e_0 \texttt{ do } e_1]\!] = \texttt{seq (while } [\![e_0]\!] \texttt{ (effect } [\![e_1]\!] \texttt{)) tup()}$$

6.1.2.4 Abstrações

O módulo '`Lang/ML/Abs`' é necessário para a definição de abstrações.

```
msos Lang/ML/Abs is
 see Lang/ML/Dec .

 see Cons/Exp, Cons/Exp/Abs, Cons/Exp/close, Cons/Exp/app-seq .
 see Cons/Abs, Cons/Abs/abs-Exp, Cons/Abs/closure .
 see Cons/Par, Cons/Par/bind, Cons/Par/tup .
 see Cons/Dec, Cons/Dec/app, Cons/Dec/rec .
sosm
```

► *Funções recursivas*

A versão de funções recursivas aqui é bastante simples e não faz uso de casamento de padrões mais complexos, já que optamos por demonstrar esta funcionalidade na semântica

da linguagem Mini-Freja, apêndice D. A nova opção para o não-terminal ⟨ dec ⟩ mostra a sintaxe de funções recursivas: o primeiro ⟨ vid ⟩ é o nome da função, o segundo é o (único) parâmetro formal e o ⟨ exp ⟩ é o corpo. Ele é convertido para a amarração do nome da função numa *closure*.

⟨ dec ⟩ → 'fun' ⟨ vid ⟩ ⟨ vid ⟩ '=' ⟨ exp ⟩

$[\![\texttt{fun i}_0\ \texttt{i}_1\ \texttt{= e}]\!] = \texttt{rec (bind } [\![\texttt{i}_0]\!] \texttt{ (close (abs (bind } [\![\texttt{i}_1]\!]\texttt{) } [\![\texttt{e}]\!]\texttt{)))}$

6.1.2.5 Concorrência

Finalmente, vamos mostrar as primitivas de concorrência em ML. Esta seção mostra as primitivas para a criação de novas *threads* e para programas ML completos.

O módulo 'Lang/ML/Conc' reúne os módulos necessários.

```
msos Lang/ML/Conc is
 see Lang/ML/Cmd', Lang/ML/Abs .

 see Cons/Cmd, Cons/Cmd/send-chan-seq,
     Cons/Cmd/start .

 see Cons/Exp, Cons/Exp/recv-chan,
     Cons/Exp/alloc-chan .

 see Cons/Sys, Cons/Sys/Cmd, Cons/Sys/conc,
     Cons/Sys/conc-chan, Cons/Sys/quiet .
sosm
```

▶ *Criação de novas* threads

A operação 'spawn' cria um novo processo, e é equivalente à construção CMSOS 'start'.

⟨ exp ⟩ → 'spawn' ⟨ exp ⟩

$$[\![\texttt{spawn } e]\!] = \texttt{seq (start } [\![e]\!]\texttt{)}$$

▶ *Programas concorrentes completos*

Todos os programas concorrentes ML devem ser prefixados por '`cml`', que é convertido para a construção CMSOS '`quiet`'.

$$\langle\, \texttt{exp} \,\rangle \rightarrow \text{`}\texttt{cml}\text{'} \langle\, \texttt{exp} \,\rangle$$

$$[\![\texttt{cml } e]\!] = \texttt{quiet (effect } [\![e]\!]\texttt{)}$$

Vamos discutir aqui a implementação concreta do analisador sintático para a conversão de ML para CMSOS. A sintaxe de ML é descrita usando uma gramática Bison e sua semântica é dada por uma série de módulos MSDF. O uso do gerador de analisadores sintáticos Bison dá-se devido a algumas limitações do interpretador Maude, discutidas no capítulo 7.

Usando as produções na gramática Bison para gerar a saída CMSOS de cada construção ML, usando a função '`format`', que é uma função similar à função C '`sprintf`', com a diferença que aloca um ponteiro e retorna a *string* criada de acordo com a formatação especificada. Por exemplo, vamos mostrar as regras para conversão de expressões condicionais em ML: o símbolo '`$$`' é, de acordo com o manual Bison, "o valor semântico do lado esquerdo da regra." Nesta especificação ele conterá a conversão de ML em CMSOS. O símbolo '`$1`' refere-se ao *token* casado pela regra.

```
exp: infexp
 | exp ANDALSO exp { $$ = format ("(cond %s %s ff)", $1, $3); }
 | exp ORELSE exp  { $$ = format ("(cond %s tt %s)", $1, $3); }
 | IF exp THEN exp ELSE exp
              { $$ = format ("(cond %s %s %s)", $2, $4, $6); };
```

Quando o analisador sintático atingir a produção de maior nível na gramática, a *string* de saída será a completa conversão CMSOS.

6.1.2.6 Exemplo

Para exemplificar esta semântica, iremos analisar um programa concorrente. Ele começa como uma única *thread* de execução que cria um canal, através da declaração '`chan`' o amarra à variável '`c`'; em seguida, ele cria três novas *threads*: a primeira envia o valor 10 para o canal '`c`', a segunda envia o valor 20 pelo mesmo canal e a terceira espera receber um valor através de '`c`'. Se buscarmos por todas as possíveis saídas deste programa, espera-se que existam dois estados finais: um em que a primeira *thread* sincronizou um sucesso e outro em que a segunda é que foi bem sucedida.

```
cml
 let chan c in
     (spawn (fn x => send (c, 10)) ;
      spawn (fn x => send (c, 20)) ;
      spawn (fn x => receive c))
 end
```

Após a conversão para CMSOS, temos o seguinte programa.

```
exec ((quiet
(effect
 (local (bind ide('c) alloc-chan)
  (seq (seq ((effect
   (seq (start (close (abs (bind (ide('x)))
        (seq (send-chan-seq ide('c) 10) (tup ())))))) (tup ()))),
   (effect (seq (start (close (abs (bind (ide('x)))
        (seq (send-chan-seq ide('c) 20) (tup ())))))) (tup ()))))
  (seq (start (close (abs (bind (ide('x)))
       (recv-chan ide('c))))) (tup ())))))) .)
```

Buscando por todas os estados finais possíveis usando o comando '`search`' de Maude, chegamos à situação esperada. A primeira solução mostra a *thread* que não sincronizou parada no ponto em que está tentando enviar o valor 10 através do canal, enquanto que a segunda solução mostra a mesma situação para o valor 20. (À medida que as *threads* terminam, elas são removidas da computação.)

```
search in CML-INTERPRETER : exec(...) =>!
```

```
    C:Conf .

Solution 1
C:Conf <- <
 quiet
  effect (
   local (ide('c)|-> chan 1)
    local (ide('x)|-> tup())
     seq send-chan-seq chan 1 10 tup())
 ::: 'Sys, {chans ={chan 1},env = void, starting' =(),
            event' = (), store =void}
>

Solution 2
C:Conf <- <
 quiet
  effect (
   local (ide('c)|-> chan 1)
    local (ide('x)|-> tup())
     seq send-chan-seq chan 1 20 tup())
 ::: 'Sys, {chans ={chan 1},env = void, starting' =(),
            event' = (), store =void}
>

No more solutions.
```

6.1.3 MiniJava

A implementação de MiniJava no *framework* de *Constructive MSOS* segue a idéia de que uma linguagem (simples) orientada a objetos é, em sua essência, uma linguagem imperativa em que classes são tipos e objetos são registros [8, 9, 54, 55]. A implementação propriamente dita é baseada em [56].

Como introdução, iremos começar com um mapeamento simples e abstrato e em seguida expandi-lo com aspectos mais avançados, tais como referência recursiva e instanciação de objetos. Uma visão bastante simples é considerá-los como registros, cujos

campos são os métodos do objeto. Podemos simplificar ainda mais usando tuplas e controlando qual método corresponde a qual projeção da tupla. Como um exemplo, vamos considerar um objeto especificado numa linguagem hipotética, similar à Java:

```
object {
  int i;
  int foo() { return i; }
  int bar() { i := i + 1; return i; }
}
```

Ele pode ser representado como uma *closure*, ou seja, uma tupla encoberta por um ambiente.

```
local (bind x 1) (tup-seq f, b)
```

Aqui, `f` e `b` são *closures* que representam, respectivamente, os métodos '`foo()`' e '`bar()`' (omitimos sua representação abstrata para simplificar a exposição). Neste mapeamento, todos os campos do objeto são declarados como amarrações que encobrem a tupla. Desta maneira, não é permitido acesso externo aos campos de um objeto diretamente; é possível, no entanto, implementar funções de acesso automaticamente para tal fim.

Para chamar um método no objeto, obtemos a projeção desejada e a computamos: por exemplo, se desejamos chamar o método '`bar()`', nós primeiro obtemos a segunda projeção, aplicando a operação '`nth(1)`' à tupla; computamos o resultado, aplicando a *closure* obtida aos seus argumentos, a tupla vazia neste exemplo.

```
app (app nth(1) o) tup()
```

onde `o` é o objeto.

Esta é uma forma bastante limitada de orientação a objetos: métodos não podem ser recursivos, nem acessar outros métodos no mesmo objeto. Iremos adicionar agora um mecanismo de auto-referência (e.g., '`this`' em Java) de forma a permitir um método chamar outros métodos no mesmo objeto. Isto é feito adicionando uma função recursiva '`self`' que, quando computada, retorna o próprio objeto. Sendo recursiva, um método pode chamar call '`self`' de dentro de si mesmo e assim ter acesso aos outros métodos do objeto, incluindo ele mesmo.

```
local
  (accum (bind x 1)
         (rec bind self
           close abs (bind tup()) (tup-seq f, g)))
  (app self tup())
```

O código acima indica que uma *closure* é amarrada recursivamente ao identificador '`self`' (esta closure não recebe argumentos, o que é especificado como '`bind tup()`' em CMSOS); o código da *closure* é, como antes, a tupla contendo os métodos. Esta *closure* é definida após a definição de '`bind x 1`' usando '`accum`'—desta maneira os campos estão disponíveis para os métodos `f` e `g`.

Nesta abordagem simples, resta apenas como instanciar um objeto baseado no tipo indicado por sua declaração de classe. Seguimos a idéia de clonagem de objetos baseados em protótipos na tradição da linguagem Self [57]. Na prática, a implementação é a seguinte: cada declaração de classe dá origem a uma declaração de função que, quando computada, retorna um novo objeto. Assim, uma classe `C` é convertida para o seguinte fragmento:

```
bind C (close abs (bind tup()) o)
```

onde `o` é um objeto-protótipo criado a partir da declaração da classe. A instanciação de objetos usa a construção `(app C tup())`, que retorna uma cópia do objeto `o`.

Com esta exposição preliminar, podemos descrever completamente o mapeamento de MiniJava para CMSOS.

A transformação da sintaxe abstrata de MiniJava em CMSOS é feita usando a mesma notação que usamos para a conversão de ML, na seção 6.1.2. A implementação foi feita usando o *framework* SableCC devido às mesmas razões que nós delineamos na descrição da semântica de ML. Como naquela seção, optamos por mostrar apenas as conversões mais relevantes, enquanto que a especificação completa encontra-se no apêndice C.

6.1.3.1 Expressões

Expressões em MiniJava consistem de operações matemáticas, identificadores, chamadas de método (que sempre retornam um valor), constantes numéricas e objetos.

⟨ exp ⟩ → ⟨ math operation ⟩ | ⟨ id ⟩ | ⟨ method invocation ⟩
| ⟨ literal ⟩ | ⟨ this ⟩ | ⟨ new ⟩

▶ *Operações matemáticas*

⟨ math operation ⟩ → ⟨ exp ⟩ ⟨ math op ⟩ ⟨ exp ⟩
⟨ math op ⟩ → '&&' | '<' | '+' | '/' | '%' | '-' | '*' | '>' | '<=' | '>=' | '='

Sejam e_i metavariáveis sobre ⟨ exp ⟩, e m sobre ⟨ math op ⟩.

$[\![e_0 \;\; m \;\; e_1]\!]$ = app $[\![m]\!]$ tup-seq ($[\![e_0]\!]$, $[\![e_1]\!]$)

▶ *Chamadas de método*

⟨ method invocation ⟩ → ⟨ exp ⟩ '.' ⟨ id ⟩ '(' ⟨ exp ⟩*')'

$[\![e \;.\; i \;(\; e* \;)]\!]$ = app (app nth(n(i)) $[\![e]\!]$) (tup-seq p)

A computação de e deve retornar um objeto; n(i) é o número do método na classe do objeto retornado por e, obtido buscando o nome do método i na meta-informação sobre a classe gerada durante a fase de análise estática; e p é construído como uma seqüência de 'ref (alloc $[\![e_i]\!]$)' que aloca uma nova entrada na memória para cada parâmetro e_i e $e*$.

▶ *Auto-referência*

⟨ this ⟩ → 'this'

$[\![$this$]\!]$ = app self tup()

▶ *Instanciação de objetos.*

⟨ new ⟩ → 'new' ⟨ id ⟩ '()'

$[\![$new i ()$]\!]$ = app i tup()

6.1.3.2 Comandos

MiniJava contém os comandos usuais encontrados em linguagens imperativas: condicionais, loops, saída, atribuição de valores, etc. Mostramos aqui apenas a conversão do comando de saída.

⟨ statement ⟩ → ⟨ if ⟩ | ⟨ while ⟩ | ⟨ block ⟩ | ⟨ print ⟩ | ⟨ assign ⟩ | ⟨ empty ⟩

▶ *Saída*

⟨ print ⟩ → 'System.out.println' '(' ⟨ exp ⟩ ')'

$[\![\texttt{System.out.println (} e \texttt{)}]\!] = \texttt{print } [\![e]\!]$

6.1.3.3 Classes

▶ *Declaração de classes*

Como descrevemos no início desta seção, uma declaração de classe define um objeto "protótipo", que é uma *closure* cujos campos tornam-se amarrações e os métodos tornam-se projeções de uma tupla.

⟨ class declaration ⟩ → 'class' ⟨ identifier ⟩ '{' (⟨ field declaration ⟩)*
(⟨ method declaration ⟩)*'}'

Sejam $\mathtt{f_i}$ metavariáveis sobre ⟨ field declaration ⟩ e $\mathtt{m_i}$ sobre ⟨ method declaration ⟩.

$[\![\texttt{class i \{ f} * \texttt{ m} * \texttt{ \}}]\!] =$

```
local
 (accum (accum [[f*]])
        (rec (bind self
                close (abs (bind dummy) tup-seq ([[m]])))))
 app self tup()
```

▶ *Declarações de métodos*

Uma declaração de métodos é convertida numa *closure* cuja lista de parâmetros formais é declarada através da construção '`tup`'. O corpo da *closure* é uma definição '`local`' com as declarações de variáveis na declaração e o corpo do método como a expressão que está sendo computada. Usamos o comando '`seq-Cmd-Exp`' de forma que a última expressão a ser computada é o valor de retorno do método.

⟨ method declaration ⟩ → ⟨ type ⟩ ⟨ identifier ⟩ '(' (⟨ parameter ⟩)*')'
'{' (⟨ var declaration ⟩)*(⟨ statement ⟩)*'return' ⟨ expression ⟩ '}'

Seja p_i uma metavariável sobre ⟨ parameter ⟩.

$[\![$ t i (p ∗) { v ∗ s ∗ return e }$]\!]$ =
close (abs tup($[\![$p∗$]\!]$) (local (accum $[\![$v∗$]\!]$) (seq (seq $[\![$s∗$]\!]$) e)))

6.1.3.4 Exemplo

Como um exemplo desta tradução, vamos definir uma classe que calcula o fatorial de um número. Esta classe implementa duas formas de cálculo: uma recursiva ('`RecFat`') e outra usando um *loop* ('`NonRecFac`').

```
class Factorial
{
  public static void main (String[] arg)
  {
    System.out.println (new Fac().Test (6));
  }
}

class Fac
{
  public int Test (int num)
  {
    int r;
    Fac recfac;
```

```
    Fac nonrecfac;
    recfac = this;
    nonrecfac = this;
    return recfac.RecFac(num) - nonrecfac.NonRecFac (num);
  }

  public int RecFac(int num)
  {
    int num_aux;
    if (num < 1) num_aux = 1 ;
    else num_aux = num * (this.RecFac(num-1));
    return num_aux;
  }

  public int NonRecFac (int num)
  {
    int i;
    int fat;
    i = num;
    fat = 1;
    while (i > 0)
      {
        fat = fat * i;
        i = i - 1;
      }
    return fat;
  }
}
```

O código convertido para CMSOS é:

```
local accum bind Fac close (abs bind @ local accum void
rec (bind self close (abs bind @ tup-seq (close (abs
tup bind num local accum bind r ref alloc 0 accum
bind recfac ref alloc 0 accum bind nonrecfac ref alloc
0 void seq seq (effect (assign-seq deref recfac app self
```

```
tup ()),effect (assign-seq deref nonrecfac app self tup
()),skip) app minus tup-seq ((app app nth(1) assigned
deref recfac tup-seq ref alloc assigned deref num),app
app nth(2) assigned deref nonrecfac tup-seq ref alloc assigned
deref num)),close (abs tup bind num local accum
bind num_aux ref alloc 0 void seq seq ((cond app lt tup-seq
(assigned deref num,1) effect (assign-seq deref num_aux
1) effect (assign-seq deref num_aux app times tup-seq (assigned
deref num,app app nth(1) app self tup () tup-seq ref
alloc (app minus tup-seq (assigned deref num,1))))),skip)
assigned deref num_aux),close (abs tup bind num local
accum bind i ref alloc 0 accum bind fat ref alloc 0 void
seq seq (effect (assign-seq deref i assigned
deref num),effect (assign-seq deref fat 1),(while app gt
tup-seq (assigned deref i,0) seq (effect (assign-seq
deref fat app times tup-seq (assigned deref fat,assigned
deref i)),effect (assign-seq deref i app minus tup-seq
(assigned deref i,1)))),skip) assigned deref fat))))
app self tup ()) void app app nth(0) local accum void rec
(bind self close (abs bind @ tup-seq close (abs tup
bind @@ local void seq seq (print (app app nth(0)
app Fac tup () tup-seq ref alloc 6),skip) 0))) app self
tup () tup-seq ref alloc 0
```

Executando em Maude, produzimos a seguinte saída:

```
rewrite in MINIJAVAPROGRAM : output(...) .
rewrites: 177289 in 58209ms cpu (58242ms real)
          (3045 rewrites/second)
result Output: output(0)
```

Descrevemos agora resumidamente a implementação. A opção de se usar SableCC, que também é um gerador de analisador sintático LALR(1) foi feita devido ao suporte presente para a transformação da sintaxe concreta na abstrata diretamente na gramática usando um estilo de reescrita, o que simplificou a construção do compilador, já que MiniJava tem uma sintaxe concreta complexa. Esta escolha também permitiu o reuso da especificação

já existente da linguagem Java 1.1, criada por Etienne Gagnon, na criaçao da gramática de MiniJava, que é uma sublinguagem de Java.

O processo foi o seguinte: primeiro, a gramática de Java 1.1 foi modificada de forma a excluir a sintaxe não suportada por MiniJava. Em seguida, uma sintaxe abstrata de MiniJava, baseada na sintaxe abstrata descrita no livro de Appel foi criada e a sintaxe concreta foi modificada para adicionar as regras de reescrita que convertem da sintaxe concreta para a abstrata. Quando este arquivo é processado, SableCC gera classes de navegação de árvore que seguem o padrão de design *visitor*. Este navegador de ávores é implementado pelo programador através da criação de uma subclasse que adiciona as ações apropriadas que devem ser executadas à medida que o navegador visita cada nó da árvore.

O processo de compilação consiste em duas fases: a análise estática, onde todos os tipos são checados e os metadados são gerados. A segunda fase visita novamente os nós da árvore, convertendo-os em seus equivalentes CMSOS. Como no caso da linguagem ML, ao invés de construir o código CMSOS em Java, podíamos simplesmente ter exportado a sintaxe abstrata e usado equações em Maude para fazer a conversão. Nesta implementação optamos por deixar o compilador MiniJava auto-contido neste sentido.

Vamos ilustrar esta descrição com um exemplo simples, que mostra como expressões aritméticas são compiladas. Começamos com a sintaxe abstrata: no fragmento abaixo o não-terminal '`expression`' contém uma regra chamada '`math_operation`'. O nome da regra é importante, já que será usada para dar nome às classes dos nós da árvore. Uma expressão aritmética é composta de duas expressões com um operador matemático, representado pelo não-terminal '`math_op`'. Os prefixos '`[lh]`' e '`[rh]`' são usados para identificar cada expressão na regra.

```
expression =
 {math_operation} [lh]:expression math_op [rh]:expression

math_op = {and} and | {lt} lt {plus} plus
        | {div} div | {minus} minus | {star} star
```

Mostramos agora a sintaxe concreta junto com as regras de transformação para a sintaxe abstrata. Para simplificar, iremos exibir apenas as regras para expressões de adição, já que, devido à resolução da precedência de expressões aritméticas, a gramática concreta tem nove regras diferentes. No fragmento abaixo, uma '`additive_expression`'

é convertida em sua forma abstrata '`expression`'. Tem três alternativas: ou é uma '`multiplicative_expression`'; ou é a opção '`plus`', onde um novo nó na árvore sintática abstrata é criado—'`math_operation`'—através da conversão recursiva de cada lado da expressão de adição; ou é a opção '`minus`', que recebe o mesmo tratamento.

```
additive_expression { -> expression }  =
    {multiplicative_expression}
        multiplicative_expression
         { -> multiplicative_expression.expression } |

    {plus}
        additive_expression plus multiplicative_expression
        { -> New expression.math_operation
                  (additive_expression.expression,
                   New math_op.plus (plus),
                   multiplicative_expression.expression) } |

    {minus}
        additive_expression minus multiplicative_expression
        { -> New expression.math_operation
                  (additive_expression.expression,
                   New math_op.minus (minus),
                   multiplicative_expression.expression) };
```

O fragmento abaixo mostra o código do navegador da árvore que passa pelo nó '`math_operation`'. O nome do método é gerado automaticamente por SableCC e recebe como parâmetro um nó que é a representação da produção '`math_operation`'. Cada componente da árvore sintática é acessado através de funções *get* e *set*. A conversão é feita mantendo-se um dicionário '`cmsos`' que mapeia cada nó em seu equivalente CMSOS. Assim, buscando as expressões '`lh`' e '`rh`' (através das funções '`n.getLH()`' e '`n.getRh()`'), obtemos a representação de cada expressão. Após termos toda a informação em mãos, simplesmente colocamos no dicionário o mapeamento CMSOS do nó corrente.

```
public
void outAMathOperationExpression (AMathOperationExpression n)
{
```

```
  String l, r;
  PMathOp o = n.getMathOp ();

  String[] args = { (String) imsos.get (n.getMathOp ()),
      (String) imsos.get (n.getLh ()),
      (String) imsos.get (n.getRh ()) };

  imsos.put (n, printf ("(app {0} tup-seq ({1}, {2}))", args));
}
```

6.2 Mini-Freja

Mini-Freja [43] é uma linguagem puramente funcional com uma semântica de computação *lazy*. Esta especificação foi feita com vários objetivos: não é baseada em CMSOS de Mosses; não precisa de ferramentas externas — a análise sintática é feita pela própria ferramenta, com algumas limitações, como iremos ver; é uma semântica *big-step*; e, finalmente, implementa casamento de padrões (*pattern matching*).

6.2.1 Sintaxe abstrata

A principal construção de Mini-Freja é a *expression*, denotada pelo conjunto '`Exp`'.

```
Exp .
Exp ::= fn Var => Exp
      | Primu Exp
      | Exp :: Exp [assoc]
      | Exp Primd Exp
      | if Exp then Exp else Exp
      | Exp Exp
      | rec Exp
      | case Exp of Rules
      | let Decls in Exp .

Exp ::= Var | Const .
```

'`Exp :: Exp`' é a sintaxe de listas, '`Exp Exp`' é a sintaxe tradicional de aplicação de expressões, '`rec Exp`' define uma expressão recursiva, '`Var`' é o conjunto de variáveis da linguagem (tecnicamente falando, são identificadores, mas estamos seguindo a nomenclatura de Pettersson), '`Const`' são as constantes, '`Primu`' são as operações primitivas *unárias* e '`Primd`' são as operações primitivas *binárias*, que são definidas da seguinte maneira:

```
Primu .
Primu ::= not | neg .

Primd .
Primd ::= lt | le | eq | ne | ge | gt | and
        | or | plus | minus | times | div | mod .

Const .
Const ::= Int | Boolean | nil .
```

Finalmente, apresentamos a sintaxe abstrata para declaração de amarrações. A sintaxe é um pouco diferente da original, devido a problemas de pré-regularidade. Ao invés de amarrar variáveis a expressões com a sintaxe '`Var = Exp`' usamos '`Var is Exp`'.

```
Decls .
Dec  .
Dec ::= Var is Exp .
Decls ::= Dec | Decls Decls [assoc] .
```

6.2.2 Semântica

Descrevemos agora a semântica dinâmica de Mini-Freja. Esta especificação é baseada nas especificações de Pettersson e Hartel, ambas em estilo *big-step*, como mencionamos, é uma linguagem funcional com computação *lazy*, ou seja, uma expressão pode ser "suspensa" e somente computada quando seu valor é realmente necessário. O único componente semântico necessário é o ambiente de amarrações.

```
Label = { env : Env, ... } .
```

Adicionamos agora o conjunto de valores à especificação, que consiste basicamente de constantes, *closures*, listas e expressões suspensas. Estas precisam de um ambiente para serem computadas no futuro.

```
Values .

Value ::= susp (Env, Exp)
        | clo (Env, Var, Exp)
        | cons (Value, Value) [assoc]
        | Const .
```

Valores são um subconjunto de expressões. Adicionamos também um novo operador de expressões, '`force e`', que "força" a computação de uma expressão suspensa `e`.

```
Exp ::= Value | force Exp .
```

Vamos começar com a computação de listas em Mini-Freja, que são seqüências de aplicações recursivas do operador '`cons`'.

```
[cons] (Exp1 :: Exp2) : Exp ={env = Env, -}=>
       cons (susp (Env, Exp1), susp (Env, Exp2)) .
```

Os operadores aritméticos são computados da maneira tradicional *big-step*.

```
           (Exp1 ={X1}=> Value1), (Exp2 ={X2}=> Value2),
                         (Value1 Primd Value2 ={X3}=> Value3)
[prim-app] -- ---------------------------------------------
           (Exp1 Primd Exp2) : Exp ={X1 ; X2 ; X3}=> Value3 .

(Int1 plus Int2) : Exp ==> (Int1 + Int2) .
(Int1 times Int2) : Exp ==> (Int1 * Int2) .
(Int1 mod Int2) : Exp ==> (Int1 rem Int2) .
(Int1 minus Int2) : Exp ==> _-_(Int1, Int2) .

(Int1 eq Int2) : Exp ==> if (Int1 == Int2) then tt else ff fi .
(Int1 ne Int2) : Exp ==> if (Int1 == Int2) then ff else tt fi .
```

A seguinte regra estabelece que formas canônicas sempre computam para si mesmo.

```
Const : Exp ==> Const .
```

Para computar a construção condicional de Mini-Freja, definimos uma operação auxiliar '`if-choose(b,`e_1`,`e_2`)`', que funciona da seguinte maneira: se `b` for verdadeiro, esta computa em e_1 e, caso contrário, para e_2.

```
Exp ::= if-choose (Value, Exp2, Exp3) .

[if-choose-tt] if-choose (tt, Exp2, Exp3) : Exp ==> Exp2 .
[if-choose-ff] if-choose (ff, Exp2, Exp3) : Exp ==> Exp3 .

     (Exp1 ==> Value),
     (if-choose (Value, Exp2, Exp3) ==> Exp),
     (Exp ==> Value')
[if] -- ------------------------------------------
     if Exp1 then Exp2 else Exp3 : Exp ==> Value' .
```

Closures computam no valore '`clo(`ρ`,`ν`,`e`)`', que consiste no ambiente "capturado" ρ, o argumento ν e a expressão e.

```
[clo] (fn Var => Exp) : Exp ={env = Env, -}=>
      clo (Env, Var, Exp) .
```

As seguintes regras especificam o significado do operador '`force`'. Essencialmente, expressões que não estão suspensas computam para elas próprias. Expressões suspensas (regra '`[force-susp]`') '`susp(`ρ`,`e`)`' são computadas substituindo o ambiente corrente por ρ e computando e em um valor intermediário ν, e finalmente obtendo o valor final ν', "forçando" a computação de ν.

```
[force-const] force Const : Exp ==> Const .

[force-clo] force clo (Env, Var, Exp) : Exp
            ==> clo (Env, Var, Exp) .

[force-cons] force cons (Value1, Value2) : Exp
        ==> cons (Value1, Value2) .

        Exp ={env = Env', -}=> Value,
```

```
                    force Value ={env = Env, -}=> Value'
[force-susp] -- ---------------------------------------------
        force susp (Env', Exp) : Exp ={env = Env, -}=> Value' .
```

Aplicação de expressões implementa um tipo de β-redução. Espera-se que '`Exp2`' na regra '`[app]`' abaixo compute para uma *closure*.

```
      Exp1 ={env = Env, -}=> clo (Env1, Var1, Exp'),
       Env2 := (Var1 |-> susp (Env, Exp2)) / Env1,
        Exp' ={env = Env2, -}=> Value
[app] -- ----------------------------------------
      (Exp1 Exp2) : Exp ={env = Env, -}=> Value .
```

A regra para variáveis segue a regra usual, com o requisito adicional que o valor retornado deve ser "forçado."

```
         Value := lookup (Var, Env),
         (force Value ={env = Env, ...}=> Value')
[lookup] -- ---------------------------------
         Var : Exp ={env = Env, ...}=> Value' .
```

A regra para o operador '`let`' computa as declarações em '`Decls`' em um conjunto de amarrações '`dec(Env')`' computa '`Exp`' sobrepondo estas amarrações ao ambiente atual.

```
Decls ::= dec (Env) .

      Decls ={env = Env, -}=> dec (Env'),
      Env'' := Env' / Env, Exp ={env = Env'', -}=> Value
[let] -- ---------------------------------------------
      (let Decls in Exp) : Exp ={env = Env, -}=> Value .
```

Para computar '`Decls`', cada '`Dec`' é computada e os ambientes resultantes são concatenados.

```
         Dec ={env = Env, -}=> dec (Env'),
         Env'' := Env' / Env,
```

```
              Decls ={env = Env'', -}=> dec(Env''')
[decls]  -- ------------------------------------
          (Dec Decls) : Decls ={env = Env, -}=>
                              dec (Env' +++ Env''') .

              Dec ==> dec (Env')
[decls] -- ------------------------
         Dec : Decls ==> dec (Env') .
```

A seguinte regra é necessária para dar a ordem de computação *lazy*: expressões não são computadas no momento em que são amarradas a variaveis; são primeiro convertidas em valores "suspensos."

```
[dec] (Var is Exp) : Dec ={env = Env, -}=>
       dec (Var |-> susp (Env, Exp)) .
```

Funções recursivas são implementadas usando um operador de ponto fixo (*fixed point operador*), seguindo idéias presentes no livro de Reynolds [8]. Espera-se que a expressão '`Exp`' seja uma expressão-lambda, tal como '`fn` ν `=>` e'.

```
               Exp (rec Exp) ==> Value
[fixed-point] -- ----------------------
               rec Exp : Exp ==> Value .
```

A seguinte regra computa uma expressão usando as construções '`exec`' e '`strict`', a última é similar à construção '`force`' com a diferença que ela opera sobre *valores*, ao passo que '`force`' opera sobre *expressões*.

```
Value ::= exec Exp | done Value .

       Exp ==> Value,
                strict Value ==> Value'
[exec] -- ----------------------------
       exec Exp : Exp ==> done Value' .

Value ::= strict Value .
```

```
[strict-const] strict Const : Value
                ==> Const .

[strict-clo] strict clo (Env, Var, Exp) : Value
             ==> clo (Env, Var, Exp) .

               strict Value1 ==> Value1',
                              strict Value2 ==> Value2'
[strict-cons] -- ------------------------------------
               strict cons (Value1, Value2) : Value
                           ==> cons (Value1', Value2') .

               Exp ={env = Env', -}=> Value,
               strict Value ={env = Env, -}=> Value'
[strict-susp] -- --------------------------------
               strict susp (Env', Exp) : Value
                           ={env = Env, -}=> Value' .
```

6.2.3 Exemplo: peneira de Eratóstenes

Vamos demonstrar a especificação de Mini-Freja implementando uma peneira de Eratóstenes usando "listas *lazy*." O algoritmo funciona da seguinte maneira: criamos uma lista infinita de números (função '`from`' abaixo). Esta lista infinita de números é filtrada de forma a manter apenas os números primos (funções '`filter`', '`sieve`' e '`not-siv`'). A partir desta lista infinita de primos, pegamos os primeiros (função '`take`' abaixo). A implementação é a seguinte. Optamos por dividir cada função em sua própria constante para simplificar a exposição. Podíamos ter criado estas constantes em um módulo MSDF, mas dado que elas são usadas apenas para simplificar a especificação e precisam de equações de qualquer maneira, optamos por declará-las em um módulo Maude.

```
ops fat n n0 xs0 x y xs pp N filter
    not-div sieve take from primes : -> Var .

op filterd : -> Dec .
op fromd : -> Dec .
```

```
op taked : -> Dec .
op not-divd : -> Dec .
op sieved : -> Dec .
op primesd : -> Dec .
op fatd : -> Dec .
```

A função '`filter`' recebe como argumentos (na forma *curry*) um predicado e uma lista e retorna apenas os elementos da lista que satisfazem o predicado. Devido à semântica do operador de ponto fixo definida formalmente no apêndice D, para declararmos uma função recursiva `f`, cujo conteúdo é uma expressão `e`, devemos usar a seguinte sintaxe: '`rec f is fn f => e`'.

```
eq filterd
= filter is rec (fn filter => fn pp => fn xs0 =>
   case xs0 of
     p nil       => nil
  || p x :: p xs => if (pp x) then
                      x :: ((filter pp) xs)
                    else
                      (filter pp) xs) .
```

A função '`from`' inicia uma lista infinita no valor especificado como seu primeiro argumento.

```
eq fromd
= from is rec (fn from =>
                (fn n => (n :: (from (n plus 1))))) .
```

A função '`take`' recebe como argumentos um número `n` e uma lista infinita `l` e retorna os primeiros `n` elementos de `l`.

```
eq taked
= take is rec (fn take => fn n0 => fn xs0 =>
   case n0 of (p 0 => nil)
            || p n => (case xs0 of
                         p x :: p xs =>
```

```
                        (x :: ((take (n minus 1)) xs))
                    || p nil       => nil)) .
```

A função '`not-div`' é usada como predicado na função '`filter`'. Ela recebe dois argumentos, `x` e `y` e retorna verdadeiro se `y` divide `x`.

```
eq not-divd = not-div is (fn x => (fn y => ((y mod x) ne 0))) .
```

A função '`sieve`' implementa a peneira removendo da lista que recebe todos os números que são divisíveis pelo resto dos números presentes na lista. A lista deve começar com o número dois, por razões óbvias.

```
eq sieved
= sieve is rec (fn sieve => fn xs0 =>
                case xs0 of
                 ((p x :: p xs) =>
                  (x :: (sieve ((filter (not-div x)) xs))))) .

eq primesd = primes is sieve (from 2) .
```

Podemos agora executar o programa reunindo todas as declarações acima e pedindo a lista dos primeiros 18 primos.

```
rewrite in PRIMES :
  < exec(let filterd taked fromd not-divd sieved primesd N is 18
         in ((take N) primes))::: 'Exp,init-rec >
result Conf :
  < done cons(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,
              59,61,nil)::: 'Exp,{env = void} >
```

6.3 Algoritmos distribuídos

Esta seção demonstra o uso do *Maude MSOS Tool* na especificação e verificação de algoritmos distribuídos. Como mencionamos anteriormente, SOS e MSOS são formalismos não somente usados na especificação linguagens de programação como também em sistemas concorrentes [4, 5]. A conversão de MSOS em lógica de reescrita feita por MMT

usando o interpretador Maude habilita o uso das ferramentas embutidas de Maude, como o verificador de modelos em lógica temporal linear e a busca em largura, detalhados na seção 2.3.1.

Esta seção está organizada da seguinte maneira: seção 6.3.1 define um modelo de execução de processos distribuídos e seção 6.3.2 mostra exemplos do livro [51] e [21]. O apêndice E contém as especificações completas de diversas variantes dos Filósofos Glutões (E.2) além de três exemplos de algoritmos distribuídos adicionais: uma exclusão mútua simples usando semáforos (E.1), o algoritmo da Padaria de Lamport (E.3) e eleição de líder (E.4).

6.3.1 Modelo de execução de processos

Esta seção descreve um modelo simples para execução de processos. Começamos com a noção de *processos* (o conjunto '`Proc`') e *identificadores de processos*.

```
Proc .
```

Um processo contém um inteiro como seu identificador (pid) e um tipo de dados abstrato que representa seu estado local ('`St`'). O estado local é dependente do algoritmo que está sendo especificado e será normalmente usado nas especificações para registrar o estado da computação de um processo, mas também pode armazenar valores temporários que são locais de um processo específico durante a execução.

```
St .
Proc ::= prc (Int, St) .
```

Seguimos idéias presentes em [58, 59] e criamos um conjunto '`Soup`' que representa uma "sopa" associativa-comutativa. Um único processo é uma sopa trivial. A evolução da sopa é feita selecionando-se não-deterministicamente um processo dentre os "processos flutuantes," através do casamento de padrão *modulo* associatividade e comutatividade, computando-se este processo, e colocando-o de volta na sopa.

```
Soup ::= Proc .
Soup ::= Soup Soup [assoc comm] .
```

A seguinte regra implementa a evolução da sopa de processos.

```
                    Proc -{...}-> Proc'
[exec1] -- ------------------------------------
        (Proc Soup) : Soup -{...}-> Proc' Soup .
```

Uma alternativa a esta regra é escrever o seu lado esquerdo como '`Soup1 Soup2`' ao invés de '`Proc Soup`'. O efeito seria a seleção não-determinística de uma porção da sopa para computação. Esta generalidade extra não é necessária em grande parte dos algoritmos presentes aqui, já que eles especificam transições para um processo específico e não um subconjunto de processos. Esta regra alternativa então seria aplicada recursivamente a ela mesma até que '`Soup1`' seja um único '`Proc`' para o qual não existem outras transições aplicáveis, gerando reescritas desnecessárias e aumentando artificialmente o espaço de estados de uma especificação.

Finalmente, necessitamos de uma regra para o caso trivial em que a sopa consiste em um único processo.

```
        Proc -{...}-> Proc'
-- -------------------------
 Proc : Soup -{...}-> Proc' .
```

6.3.1.1 Modelos de comunicação de processos

Esta seção descreve dois possíveis modelos de comunicação de processos: memória compartilhada e troca de mensagens.

Memória compartilhada é trivialmente implementada com o uso de um componente *read-write* no rótulo para armazenar as variáveis compartilhadas pelos processos. O restante desta seção mostra como implementar um modelo simples de troca de mensagens em uma rede assíncrona.

O conjunto '`Msg`' representa as mensagens que circularão na rede.

```
Msg .
```

O formato da mensagem é, como é usual, dependente do algoritmo, mas, para esta exposição vamos assumir o seguinte:

```
Msg ::= msg Int from Int to Int .
```

onde o primeiro argumento é o valor a ser transportado pela mensagem, o segundo é a origem da mensagem e o terceiro o destino.

O mecanismo de troca de mensagem segue as funcionalidades do casamento de padrões de Maude. Assim, mensagens e processos "flutuam" na sopa e as regras de transição simulam a transmissão de uma mensagem a um processo através do casamento do argumento que representa o destino de mensagem com o pid presente no processo destinatário. Para isto, precisamos expandir o conjunto '`Soup`' para permitir também a presença de mensagens.

```
Soup ::= Msg | Proc .
```

Para exemplificar a troca de mensagens através de casamento de padrões, considere-se o seguinte fragmento em que uma mensagem originada do processo '`Int'`' com destino o processo '`Int`' é pareada com um processo cujo pid é '`Int`'.

```
prc (Int, St) (msg C from Int' to Int)
```

Dado que agora processos e mensagens precisam interagir para a evolução da sopa, precisamos generalizar a regra de *interleaving* para permitir a evolução de uma porção da sopa.

```
                Soup1 -{...}-> Soup'1
 [exec2] -- ---------------------------------------
         (Soup1 Soup2) : Soup -{...}-> Soup'1 Soup2 .
```

Esta regra tem as desvantagens descritas na seção 6.3.1, mas sua generalidade permite seu uso em uma grande variedade de diferentes interações entre processos e mensagens, adaptando-se às necessidades das diversas especificações.

6.3.1.2 Justiça

Vamos discutir sobre a falta de justiça na regra '`[exec1]`'. É fácil notar que não existe ordem específica em que processos são selecionados para serem computados: todos os traços possíveis de execução são produzidos, incluindo aqueles em que um processo específico executa continuamento, impedindo qualquer outro processo de evoluir.

Uma maneira bastante simples para adicionar justiça a esta especificação é controlando qual processo é escolhido para ser computado através de algum mecanismo de *escalonamento*. Vamos descrever um destes mecanismos, o mecanismo *justo* (ou *round-robin*) de escalonamento de processos. Ele consiste em um contador que opera *modulo* o número de processos presentes: o valor corrente do contador é o pid do processo que deve ser executado em seguida; após executar um passo, o contador é incrementado. Com esta estratégia, todos os processos eventualmente atingem a sua vez de execução.

Isto é implementado adicionando-se um componente *read-write*, indexado por '`fair`' ao rótulo.

```
Label = { fair : Int, fair' : Int, ... } .
```

Mudamos agora a regra '`[exec1]`' para refletir o escalonamento descrito. Vamos assumir que existe uma constante '`n`' (a ser instanciada mais tarde através de uma equação) que contém o número de processos na sopa.

```
Int' := (Int + 1) rem n,
prc (Int, St) -{fair = Int, fair' = Int, ...}-> prc (Int, St')
-- ------------------------------------------------------
(prc (Int, St) Soup) : Soup -{fair = Int, fair' = Int', ...}->
                                          prc (Int, St') Soup .
```

Esta solução, apesar de funcionar, é bastante restrita: todos os processos recebem a mesma probabilidade de execução, o que não é o caso sempre, e os processos executam sempre na mesma ordem. Esta última restrição pode ser eliminada usando um gerador de números pseudo-aleatórios e escolhendo aleatoriamente qual processo a ser executado.

6.3.2 Exemplos

6.3.2.1 Outro jogo de *threads*

Vamos começar com uma especificação simples, baseado no *thread game* descrito em [21]. Esta especificação também demonstra os problemas associados com a (falta de) justiça na especificação.

Duas *threads* continuamente tentam atualizar o valor de uma variável compartilhada: um processo incrementa o valor em um e o outro diminiu o valor de um. Esta variável compartilhada é modelada usando um componente *read-write* indexado por '`v`'.

```
Label = {v : Int, v' : Int, ...} .
```

Vamos formalizar o comportamento de ambas as *threads*. Processo '`prc 0`' incrementa e o processo '`prc 1`' diminui. Vamos também limitar o valor da variável compartilhada para ser entre zero e cinco (um valor arbitrário).

```
   Int < 5, Int' := Int + 1
-- ------------------------------------------------
(prc 0) : Proc -{v = Int, sh' = Int', -}-> prc 0 .

   Int > 0, Int' := Int - 1
-- ------------------------------------------------
(prc 1) : Proc -{v = Int, sh' = Int', -}-> prc 1 .
```

As duas regras seguintes mantêm o sistema rodando quando a variável está nos limites estabelecidos.

```
   Int >= 5, Int' := Int
-- ------------------------------------------------
(prc 0) : Proc -{v = Int, sh' = Int', -}-> prc 0 .

   Int <= 0, Int' := Int
-- ------------------------------------------------
(prc 1) : Proc -{v = Int, sh' = Int', -}-> prc 1 .
```

Para analisar esta especificação com o verificador de modelos de Maud (seção 2.3.1.3), vamos criar uma proposição '`max(i)`', que é verdade sempre quando a variável for igual ou inferior a `i`.

```
op max : Int -> Prop .

ceq (< S, { v = I', PR } >) |= max (I) = true
if I' <= I .
```

Se usarmos o escalonamento justo de processos descrito na seção 6.3.1.2, verificamos que o valor da variável compartilhada nunca excede um.

```
rewrites: 2511 in 26ms cpu (26ms real) (93013 rewrites/second)
reduce in CHECK :
  modelCheck(init,[] max(1))
result Bool :
  true
```

Se retirarmos a justiça, o verificador de modelos rapidamente exibe um contra-exemplo, onde o processo zero sempre incrementa a variável compartilhada até o limite de cinco.

```
reduce in CHECK :
  modelCheck (init, [] max(1))
result ModelCheckResult :
  counterexample(
  { < (prc 0 prc 1), {fair = 0, v = 0} > }
  { < (prc 0 prc 1), {fair = 0, v = 1} > }
  { < (prc 0 prc 1), {fair = 0, v = 2} > }
  { < (prc 0 prc 1), {fair = 0, v = 3} > }
  { < (prc 0 prc 1), {fair = 0, v = 4} > },
  { < (prc 0 prc 1), {fair = 0, v = 5} > })
```

6.3.2.2 Filósofos glutões

Esta seção exibe uma solução do problema dos "Filósofos Glutões" de Dijkstra, descrita em [51]. Esta solução é baseada na quebra da simetria no momento em que cada filósofo pega seu garfo: filósofos (representados por processos) com pids pares pedem inicialmente o garfo à sua esquerda enquanto que filósofos com pids ímpares pedem inicialmente o garfo à sua direita.

Por definição, o garfo à direita de um filósofo $\mathtt{i}$ tem número $\mathtt{i}$ e o garfo à esquerda tem número $\mathtt{i}+\mathtt{1}$ mod $\mathtt{n}$. Quando há uma competição para pegar um garfo, os pids dos filósofos em competição são inseridos numa fila presente em cada garfo. À medida que cada filósofo termina de comer, ele remove o seu pid da fila de seus garfos.

A especificação MSDF é a seguinte. Primeiro, precisamos mapear cada identificador de garfo a uma lista de pids para implementar suas filas. O conjunto '`Pids`' define a lista de pids e '`Queue`' define o mapa de inteiros (identificadores de garfo) para '`Pids`'.

Apesar de cada fila ser compartilhada apenas entre dois filósofos, para simplificar a especificação optamos por fazê-las globalmente acessíveis através de um componente *read-write* indexado por '`q`'.

```
Pids = (Int) List .
Queue = (Int, Pids) Map .
Label = {q : Queue, q' : Queue, ...} .
```

A especificação é parametrizada por uma constante '`n`' que deve ser instanciada através de uma equação para o número correto de filósofos à mesa.

```
Int ::= n .
```

Cada filósofo é um processo com os seguintes estados. Cada estado será detalhado nas transições exibidas abaixo.

```
St .
St ::= srem
     | stest-right
     | stest-left
     | sleave-try
     | scrit
     | sreset-right
     | sreset-left
     | sleave-exit
     | stry
     | sexit .
```

Iremos mostrar apenas as transições para processos ímpares. As transições para processos pares são simétricas às exibidas aqui. Inicialmente, todos os filósofos estão com fome e irão tentar pegar seus garfos, ou seja, todos os processos estão no estado '`stry`'. Processos ímpares, selecionados com o predicado '`odd(i)`', tentam pegar seus garfos à direita (estado '`stest-right`').

```
                    odd (Int)
-- ---------------------------------------------
prc (Int, stry) : Proc --> prc (Int, stest-right) .
```

Neste ponto fazemos uma pequena modificação no algoritmo original. A regra original é a seguinte: se o garfo não está disponível, o processo coloca seu pid na fila, e volta a estar se o seu pid chegou ao início da fila, como mostra a regra abaixo. Lembre-se (seção 4.3) que '`insert-back`' e '`first`' são funções internas que operam em listas parametrizadas.

```
odd (Int),
Pids := lookup (Int, Queue),
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids) else Pids fi,
Queue' := (Int |-> Pids') / Queue,
St := if first (Pids') == Int
         then stest-left else stest-right fi
-- ---------------------------------------------
prc (Int, stest-right) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, St) .
```

Esta *espera ocupada* (*busy waiting*) torna a verificação mais complexa, já que, se o algoritmo não estiver correto, ele entrará em *livelock* e não em *deadlock*. *Deadlocks* são mais simples de serem descobertos em Maude: o verificador precisa buscar por um estado ao qual nenhuma regra se aplica, já que o sistema é reativo. Optamos por modificar o algoritmo de forma a permitir no máximo um processo na fila, tornando-a na verdade num semáforo. A transição irá apenas ocorrer quando um processo conseguir um garfo através da inserção de seu pid na fila e a verificação de que o seu pid é o primeiro—ou seja, a fila estava vazia. Se o garfo for adquirido com sucesso, o processo passa a pegar o garfo à sua esquerda mudando o seu estado para '`stest-left`', caso contrário, ele não muda de estado.

```
odd (Int),
Pids := lookup (Int, Queue),
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids) else Pids fi,
Queue' := (Int |-> Pids') / Queue,   first (Pids') == Int
-- ---------------------------------------------------
prc (Int, stest-right) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, stest-left) .
```

Esta regra não é apenas mais simples que a anterior, também tem a vantagem de criar um *deadlock* ao invés de um *livelock* se a especificação tiver algum problema.

O restante da especificação encontra-se no apêndice E.2: após pegar o seu garfo à direita, o processo tenta pegar o garfo à esquerda usando uma transição similar. De posse dos dois garfos, o processo move para sua região crítica e passa a se desfazer dos garfos: primeiro o à direita e depois o à esquerda. Finalmente, entra no estado '`srem`', que representa o filósofo pensando. O processo move do estado '`srem`' diretamente para '`stry`', indicando que logo após pensar, um filósofo volta a ficar com fome novamente.

Buscando por um estado final com o comando '`search`' e a relaçaõ '`=>!`' é uma boa maneira para procurar por um *deadlock* no algoritmo, já que um estado final é um estado ao qual nenhuma regra se aplica, significando que o conjunto de processos está parado e não pode mais evoluir.

A função axiliar '`initial-conf`' cria uma configuração inicial com o número desejado $\mathtt{n}$ de filósofos. Para $\mathtt{n} = 4$, o algoritmo leva 3.6 para descobrir que não há estado final, como esperamos numa especificação correta.

```
rewrites: 760825 in 3604ms cpu (3646ms real)
          (211079 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

Quando $\mathtt{n} = 6$, a busca leva dois minutos.

```
rewrites: 26197002 in 127450ms cpu (127420ms real)
          (205547 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

Podemos testar a especificação com mais buscas. Por exemplo, sabemos que, numa configuração com quatro filósofos, dois filósofos podem comer ao mesmo tempo (ou seja, estar em seus respectivos estados '`scrit`'), mas um filósofo nunca pode comer junto com seu vizinho. Podemos verificar isto com uma busca que retorna todas as configurações em que dois filósofos estejam em seus estados '`scrit`':

```
search in SEARCH : initial-conf =>*
 < (prc(I1:Int,scrit)prc(I2:Int,scrit) S:Soup)::: 'Soup,
   R:Record > .

I1:Int <- 0 ; I2:Int <- 2
I1:Int <- 1 ; I2:Int <- 3
I1:Int <- 2 ; I2:Int <- 0
I1:Int <- 3 ; I2:Int <- 1
```

Outra busca confirma que três filósofos nunca comem ao mesmo tempo em uma configuração com quatro filósofos.

```
search in SEARCH : initial-conf =>*
 <(prc(I1:Int,scrit)prc(I2:Int,scrit)prc(I3:Int,scrit)
   S:Soup)::: 'Soup,R:Record > .

No solution.
```

Devido à falta de justiça, é possível que um filósofo específico nunca consiga comer, como a seguinte verificação de modelos mostra. A proposição '`state(i,s)`' é verdade quando processo `i` está no estado `s`. Olhando no contra-exemplo, vemos que o processo '`0`' está "preso" em '`sleave-try`' enquanto processo '`2`' fica continuamente entrando e saindo de sua região crítica.

```
rewrites: 3539 in 60ms cpu (60ms real) (58983 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(0,scrit))
result ModelCheckResult :
  counterexample(
  { prc(0,stry) prc(1,stry) prc(2,stry) prc(3,stry)}...,

  { prc(0,sleave-try) prc(2,srem) ... }
  { prc(0,sleave-try) prc(2,stry) ... }
  { prc(0,sleave-try) prc(2,stest-left) ... }
  { prc(0,sleave-try) prc(2,stest-right) ... }
  { prc(0,sleave-try) prc(2,sleave-try) ... }
```

```
{ prc(0,sleave-try) prc(2,scrit) ... }
{ prc(0,sleave-try) prc(2,sexit) ... }
{ prc(0,sleave-try) prc(2,sreset-left) ... }
{ prc(0,sleave-try) prc(2,sreset-right) ... }
{ prc(0,sleave-try) prc(2,sleave-exit) ... }
```

O apêndice E.2 completa esta seção com outras variações na solução dos Filósofos Glutões, como uma especificação que cada filósofo pára após comer, e uma especificação com escalonamento justo de processos. Também produzimos uma especificação *errada* e usamos as ferramentas disponíveis em Maude para detectar o *deadlock*.

Capítulo 7

Conclusão

Este capítulo mostra as contribuições deste trabalho e discute algumas limitações da implementação atual de MMT, além de mostrar algumas linhas possíveis de trabalhos futuros. Está organizado da seguinte maneira: a seção 7.1 discute algumas limitações da implementação de MMT; a seção 7.2 discute possíveis melhorias à ferramenta e a seção 7.3 mostra as constribuições do trabalho.

7.1 Decisões de implementação e limitações

A seção 7.1.1 explica a escolha de usar um conjunto mínimo padrão nas árvores sintáticas tipadas que aparecem nas transições em condições MSDF; a seção 7.1.2 descreve alguns problemas na análise sintática de módulos MSDF que tornam a linguagem um pouco diferente da proposta de Mosses; a seção 7.1.3 explica um efeito-colateral da nossa escolha do processo de compilação; a seção 7.1.4 discute aspectos de pré-regularidade em relação a especificações MSDF; finalmente, a seção 7.1.5 discute a tipagem automática de metavariáveis.

7.1.1 Árvores sintáticas tipadas nas condições

O uso de um conjunto mínimo como tipo padrão da árvore sintática em transições que aparecem nas condições de regras MSDF (seção 4.2.4) poderia ser generalizado de forma a permitir a especificação do tipo das árvores sintáticas da mesma maneira que ocorre nas conclusões. Contudo, a proposta atual parece ser suficiente, sob um ponto de vista pragmático, dado o número de exemplos que foram feitos. Talvez a razão seja pois, em semântica operacional estrutural, é comum a condição ser sobre uma subárvore muito específica da árvore na conclusão, que normalmente tem o menor conjunto possível como

tipo.

7.1.2 Limitações da sintaxe de MSDF em MMT

O analisador léxico e sintático de Maude 2.1.1 permite a especificação de uma sintaxe bastante flexível, mas poucos símbolos de pontuação são usados para separar identificadores, forçando o usuário a inserir espaços para evitar erros de análise léxica. Por exemplo, é bastante comum em linguagens a declaração de uma sintaxe de atribuição tal como '`_:=_`', mas não é possível escrever '`x:=y`', já que seria identificado como um único símbolo interpretado por uma "constante." O mesmo problema não acontece quando os símbolos '`{`', '`}`', '`,`', '`(`', '`)`', dentre outros, são usados: é possível escrever algo como '`{({z,(x,y)})}`' sem a necessidade de inserção de espaços.

Outro problema é que, atualmente, não há maneira para especificar a "forma" de alguns *tokens* (por exemplo, sua primeira letra deve ser em maiúsculas). Uma solução seria a adição de um gerador de analisador léxico em Maude que combina a flexibilidade de expressões regulares com a semântica algébrica de operadores Maude.

Com algum esforço de programação, estas restrições podem ser removidas devido à natureza reflexiva de Maude. Para lidar com os requisitos de espaçamento, pode-se criar uma função de "filtro" com assinatura `QidList` $\rightarrow$ `QidList`, que processa a entrada de *tokens* vindos do '`LOOP-MODE`' "quebrando" os tokens em determinados caracteres específicos. Pr exemplo, uma entrada como '`x:=y`' seria transformada pelo '`LOOP-MODE`' em '`'x:=y`' e a função quebraria em '`'x ':= 'y`', de acordo com alguma regra que torna qualquer seqüência de pontuação um separador de *tokens*. Uma solução no meta-nível também é aplicável para lidar com as limitações de análise léxica devido à falta de expressões regulares: após entrar o texto e seu parsing em um tipo de dados, pode-se criar uma função que faz uma checagem de tipos no meta-nível que rejeita programas mal formados, efetivamente *programando* o que uma expressão regular especificaria.

Ainda, quando uma gramática concreta de uma linguagem de programação faz uso de módulos internos, conflitos entre operadores predefinidos e os existentes na gramática normalmente acontecem, como o exemplo abaixo mostra:

```
fmod PROBLEM1 is
 inc INT .

 sorts Value Exp Id .
```

```
 subsort Value < Exp .
 subsort Id < Exp .
 subsort Int < Value .

 op _+_ : Exp Exp -> Exp [ditto] .
 op _-_ : Exp Exp -> Exp [ditto] .
 op _<_ : Exp Exp -> Exp .
endfm
```

Lendo este módulo em Maude, recebemos o seguinte aviso:

```
Warning: "limitations.maude", line 23 (fmod PROBLEM1):
    declaration for _<_ has the same domain kinds as the
    declaration on "prelude.maude", line 190 (fmod NAT)
    but a different range kind.
```

De fato, no módulo '`NAT`' (predefinido), o operador '`_<_`' tem imagem o *sort* '`Bool`', desconexo do *sort* de imagem '`Exp`' de '`_<_`'. Também, para suprimir outro aviso do interpretador de Maude, tivemos que usar o atributo '`ditto`' nas operações '`_+_`' e '`_-_`', o que indica que os atributos destas operações devem ser os mesmos das operações que foram definidas para os *sorts* relacionados a '`Exp`', tais como '`Nat`', '`Int`', etc., ou seja, '`[assoc comm prec 33]`'. Aparentemente, não há uma boa solução para isto, além de modificar as operações predefinidas para usar nomes menos "naturais", tal como '`ADD`' ao invés de '`_+_`'. Esta solução quebraria a compatibilidade com todas as especificações escritas até o momento para Maude e realmente não é a recomendada. Uma solução usual é isolar os *sorts* predefinidos, como '`Int`' usando funções de coerção para *sorts* artificiais (por exemplo, '`Integers`'). A solução mais simples é, evidentmente, não usar os símbolos que geram o conflito.

Todas estas restrições são responsáveis por uma diferença entre o MSOS Tool de Mosses e *Maude MSOS Tool*, além do uso de Bison e SableCC para definir formalmente a sintaxe das linguagens de programação ML e MiniJava. Algumas das diferenças entre a ferramenta de Mosses e MMT são: (i) a linha de hífens que separa a conclusão das premissas deve ser necessariamente começar com '`--`' seguido de um espaço. Em maude três hífens ('`---`') iniciam um comentário de linha; (ii) restrições no espaço em relação a declarações de rótulos e árvores sintáticas tipadas.

7.1.3 Carregamento de módulos

Maude MSOS Tool segue Full Maude na forma como este lida com hierarquia de módulos: os módulos incluídos são incorporados no módulo final, compilado, chamado de um "módulo achatado" (*flat module*). Este esquema tem como conseqüência um aumento no tempo de carga de módulos, como acontece no caso da especificação de CMSOS (da ordem de 100 módulos), já que, à medida que módulos incluem outros móudlos, suas versões achatadas (metarrepresentadas no banco de dados de Full Maude) crescem cada vez mais. Isto também acontece em Full Maude pela mesma razão e um experimento simples confirma isto. Gerando 100 módulos aleatórios, com inclusões aleatórias entre eles, mas com o mesmo conteúdo (sem sentido) abaixo,

```
(omod name98 is
  including name47 .
  including name10 .
  including name79 .

  sort s98 .

  ops c98 d98 : -> s98 .
  ops x98 y98 : s98 s98 -> s98 .

  eq x98(c98,d98) = d98 .

  vars W WW : s98 .

  crl y98(W,WW) => d98 if W => WW .
endom)
```

obtemos os seguintes números: o primeiro módulo lido necessita de 2000 reescritas e 240 milisegundos para ser processado por Full Maude. Os módulos finais, dependendo do número de inclusões que eles carregam (não mais do que dez são geradas por módulo) podem levar até *26 milhões de reescritas* e *90 segundos* para serem lidos. Usamos módulos "orientados a objeto" ('`omod ... endom`') para o teste pois, em Full Maude, eles também passam por um processo de compilação.

O exemplo mostrou uma especificação extremamente conexa. Mas uma que seja moderadamente conexa, como a CMSOS (seção 6.1, apêndice A) em MMT também pode ter problemas. Por exemplo, o módulo que lê todos os módulos CMSOS necessários para dar a semântica à linguagem ML (seção 6.1.2) necessita de 950000 reescritas e 16 segundos para ser lido.

O interpretador Maude não tem este problema, como testes similares comprovam. Infelizmente, Maude 2.1.1 não tem a capacidade de ler um módulo metarrepresentado e, se houvesse a necessidade de MMT ser implementado em Maude diretamente ao invés de Full Maude por motivos de eficiência, ele perderia a capacidade de, numa *única sessão* ler e executar especificações MSDF. Funcionaria apenas como um "preprocessador" que, na primeira execução, emite código Maude a partir de especificações MSDF para um arquivo e, numa segunda execução, lê este arquivo para executar (ou verificar) a especificação convertida. Toda esta complexidade pode ser escondida usando-se um arquivo de *script*, mas, neste ponto em particular, a ferramenta deixaria de ser completamente formal.

7.1.4 Limitações da generalidade de MSDF em MMT

Módulos na lógica equacional de pertinência devem ser pré-regulares [13], ou seja, cada termo `t` deve ter um *sort mínimo*. Este requisito algébrico pode afetar a modularidade, como o seguinte exemplo, presente em [60], mostra. Os módulos '`SIG0`' e '`SIG2`' são pré-regulares.

```
fmod SIG0 is
 sorts t1 t2 .
endfm

fmod SIG2 is
 including SIG0 .
 op f : t1 -> t1 .
 op f : t2 -> t2 .
endfm
```

Se combinarmos estes módulos com '`SIG`' exibido abaixo, a operação '`f`' perde a sua pré-regularidade e ainda recebemos um alerta do interpretador Maude.

```
fmod SIG is
```

```
 including SIG2 .
 sort s .
 subsort s < t1 .
 subsort s < t2 .
endfm

Advisory: "pré-regular-combination.maude", line 11 (fmod SIG):
    operator f has been imported from both
    "pré-regular-combination.maude", line 7 (fmod SIG2) and
    "pré-regular-combination.maude", line 8 (fmod SIG2) with
    no common ancestor.
Warning: sort declarations for operator f failed
    pré-regularity check.
```

À medida que especificações se tornam complexas, principalmente no caso de uma especificação bastante modular como CMSOS, com um grande número de módulos, conjuntos e relações de inclusão que são diretamente mapeados para *sorts* e *subsorts*, a possibilidade de incluir um módulo que resulte na perta da pré-regularidade não deve ser ignorada. Este fato é, infelizmente, uma característica da lógica equacional de pertinência e não de MMT. O que a ferramenta atualmente necessita é de alguma forma de avisar o usuário sobre módulos não pré-regulares. Atualmente isto é feito pela ferramenta Maude somente após a compilação, o que pode confundir novos usuários que não têm conhecimento sobre a implementação.

Outra melhoria que tornaria MMT mais compatível com a ferramenta MSOS Tool de Mosses é a capacidade de ler módulos dinamicamente. Atualmente, a leitura de módulos através dos comandos '`in`' ou '`load`' não é parte do formalismo propriamente dito (da mesma maneira que o comando '`rewrite`' é, por exemplo). Apesar de MMT incluir automaticamente módulos que satisfazem determinados requisitos (seção 4.2.1), eles devem estar previamente carregados em Full Maude para serem efetivamente usados. Uma conseqüência disto é que todos os módulos devem ser lidos em uma ordem específica diretamente relacionada à sua relação de dependência. Uma maneira de evitar este passo trabalhoso é a leitura dinâmica de módulos necessários, caso eles ainda não tenham sido carregados. A ferramenta de Mosses usa o comando '`ensure_loaded`' de Prolog para obter este efeito.

Finalmente, o `LOOP-MODE` trata os arquivos carregados pelo usuário como uma fila infinita de *tokens*, o que impede que os arquivos sejam unidades de compilação. Isto faz com que os módulos em Full Maude sejam delimitados, como '`msos ... sosm`', '`tmod ... endtm`', '`omod ... endom`', etc. Esta é a última diferença entre MMT e MSOS Tool, em que arquivos são módulos e a hierarquia de diretórios é usada para nomeá-los.

7.1.5 Metavariáveis automáticas

A criação automática de metavariáveis cujo tipo é baseado em seus nomes claramente limita o uso de nomes de metavariáveis em especificações MSDF. Isto pode ter um efeito negativo em regras que necessitam de diversas metavariáveis do mesmo tipo, tal como a seguinte, presente no apêndice E.1, onde '`Int`' e '`Int'`' têm o mesmo nome, mas correspondem a objetos distintos: '`Int`' refere-se ao pid e '`Int'`' é o valor do semáforo.

```
                          Int' == 0
 -- ---------------------------------------------------
 prc (Int, down) : Proc -{sem = Int', sem' = Int', -}->
 prc (Int, down) .
```

Isto é, evidentemente, uma questão de engenharia de software. Uma solução possível para a regra acima é criar um outro conjunto que seja um superconjunto dos inteiros e usar este nome de conjunto como metavariável ao invés de '`Int'`'. Este problema não ocorre nas especificações de CMSOS e Mini-Freja.

Acreditamos que um algoritmo de inferência de tipos não funcionaria no contexto de especificações ordenadas-sortidas, apesar de que não iremos provar esta afirmação. Esperamos que o seguinte exemplo torne o problema claro:

```
Exp .
Exp ::= Int | add(Exp, Exp) .

             A -{...}-> A'
--- ------------------------------
add(A,B) : Exp -{...}-> add(A',B) .

             D -{...}-> D'
--- ------------------------------
```

```
add(C,D) : Exp -{...}-> add(C,D') .

add(E,F) --> E + F .
```

Um algoritmo de inferência correto deveria determinar que o tipo das variáveis '`A`'–'`F`' fossem, respectivamente, '`Exp`', '`Exp`', '`Int`', '`Exp`', '`Int`' e '`Int`'. As únicas duas variáveis cujo tipo pode ser seguramente determinado são '`E`' e '`F`', isto porque '`_+_`' só está definido para '`Int`'. Caso contrário, devido à inclusão do conjunto '`Int`' em '`Exp`', não é possível saber, por exemplo, que '`C`' deva ser '`Int`' e não '`Exp`'. Uma solução por força bruta com a adição de todas as combinações possíveis de '`Exp`' e '`Int`' para as outras regras funcionaria (testes mostram isto), mas geraria um número desnecessário de reescritas. A combinação de inclusão de conjuntos e metavariáveis poderia ser exponencial numa especificação grande.

7.2 Melhorias possíveis da ferramenta — trabalhos futuros

A relação entre MSOS e MRS desenvolvida em [18, 20] usa regras de reescrita condicionais para a semântica de transições MSOS, já que estas cobrem o caso genérico de sistemas de transição não-determinísticos e não-terminantes. Contudo, para um fragmento determinístico e terminante de um sistema de transição, poder-se-ia usar *equações* ao invés de regras e manter estas apenas no caso em que não-determinismo e/ou não-terminação poderia ocorrer. Por exemplo, a regra '`[let1]`' na página 53 poderia ser convertida em uma equação condicional tal como:

```
ceq [let1] : { let X = I in E end, {(env = Env), PR} }
    = [ let X = I in E' end, {(env = Env), PR'} ]
if Env' := override (Env, X, I) /\
   [ E', {(env = Env'), PR'} ] := { E, {(env = Env'), PR} } .
```

Infelizmente a combinação de regras e equações provavelmente implicará dificuldades em sua interação. Considere regras MSDF para processos concorrentes, atribuição de variáveis e comandos de *loop* que provavelmente seriam convertidos em regras que têm reescritas nas condições, como, por exemplo:

```
crl { E1 || E2, R } => [ E'1 || E2, R' ]
```

```
if { E1, R } => [ E'1, R' ] .
```

onde '`_||_`' é uma sopa associativa-comutativa de expressões. Não haverá equação que case com a condição (já que são igualdades e não reescritas). Devemos mencionar que, no caso de uma especificação de uma linguagem de programação concorrente, uma solução comum em Lógica de Reescrita adaptado ao *framework* MRS é usar um *conjunto de configurações*, ao invés de uma única configuração [34]. Cada configuração teria o seu próprio texto de programa e registro semântico; a execução concorrente seria dada por regras de reescrita sobre este conjunto e cada configuração seria reescrita usando equações para a parte determinística e terminante, e regras para a parte não-determinística e não-terminante. Esta maneira de especificar abre a possibilidade de se adicionar *concorrência real* para especificações MSDF, ao invés da semântica atual de concorrência *interleaving*. Isto provavelmente necessitaria de algum estudo em novos tipos de categorias de rótulos para suportar traços verdadeiramente concorrentes de execução face aos requisitos de composicionalidade de MSOS.

Outra limitação é que a verificação de modelos de especificações MSDF com a conversão atual que usa regras de reescrita condicionais é problemática já que as reescritas nas condições na lógica de reescrita são "reescritas descartáveis" [12]. Assim, estados que ocorrem apenas nas condições não são acessíveis para consultas ao verificador de modelos em fórmulas LTL. Por exemplo, nas transições principais, o ambiente de amarrações é sempre constante, mantendo sempre o seu valor inicial, enquanto que as amarrações dos identificadores propriamente ditas ocorrem nas condições. Com esta limitação, as consultas ao verificador de modelos devem ser feitas através da observação em componentes apenas-escrita ou escrita-leitura, como a memória, ou explorando alguma propriedade que envolva todo o texto do programa e não apenas alguma parte. Além disto, reescritas condicionais normalmente diminuem a eficiência do processo de reescrita [61]. Uma possível solução para testes problemas é usar apenas *reescritas incondicionais*, seguindo as idéias presentes em *evaluation contexts* [62] e o uso de *estratégias de reescrita* [26] para remover a necessidade do uso da regra qtstep (seção 2.4).

A combinação de *evaluation contexts* e equações incondicionais pode levar a um aumento significativo na eficiência, mas, acreditamos, ao custo de uma diminiuição significativa na legibilidade, de acordo com um protótipo que desenvolvemos usando estas técnicas para um subconjunto de Concurrent ML (`http://www.ic.uff.br/~cbraga/losd/specs/cml-cps/cml.maude`) como visto em [63] e [34] mostra—mais um incentivo para adicionar estas funcionalidades ao MMT.

Outra possível melhoria pode vir de idéias da área de *partial evaluation*. É possível que um programa P não faça uso de todo o conjunto de transições presentes em uma especificação $\mathcal{S}$. A partir desta constatação, pode-se gerar uma "especificação especializada" $\mathcal{S}_{\mathsf{P}}$ que contém apenas as construções necessárias para a execução e verificação de P. Se o conjunto de declarações omitido for grande, o casamento de termos a regras provavelmente será feito bem mais rapidamente por Maude. Quando Maude tiver um compilador de teorias de reescritas, será possível efectivamente compilar um programa P através da criação da especificação especializada $\mathcal{S}_{\mathsf{P}}$ e compilando esta especificação, usando o compilador de Maude.

A ferramenta necessita de uma melhor interface com o usuário. Atualmente, os erros aparecem em três diferentes níveis: aqueles que o MMT reporta, os erros que são exibidos por Full Maude, e aqueles que são exibidos apenas pelo interpretador Maude. Esta situação é bastante confusa, mas uma solução mais robusta precisaria da adição a MMT de um conhecimento signifificativo sobre MEL para prever qualquer problema que um módulo específico poderá ter após a compilação. Da mesma maneira que a função '`metaParse`' de Maude retorna um erro indicando um problema de análise sintática, seria ideal a presença de outros comandos no meta-nível para a análise de metamódulos e, de maneira similar, listar problemas.

Finalmente, seguindo o exemplo de LETOS (capítulo 3), uma opção de saída LaTeX e alguma maneira de obter o traço de execução que seja mais perto do domínio MSOS do que o Maude traria um grande impacto na facilidade de uso da ferramenta.

7.3 Contribuição

As principais contribuições de nosso trabalho foram: (i) desenvolver um interpretador MSOS que usa uma linguagem de especificação mais adequada ao domínio de especificações MSOS do que especificações Maude. Esta característica permite a possibilidade de formatar especificações MSOS que são bem mais similares à notação matemática de MSOS (e SOS); (ii) implementar uma nova conversão de MSOS para lógica de reescrita, baseada no novo trabalho de Braga e Meseguer em [25, 20]; (iii) demonstrar a usabilidade da ferramenta e do *framework* CMSOS através do desenvolvimento de diferentes especificações de linguagens de programação; (iv) demonstrar o que pode ser obtido quando desenvolve-se uma ferramenta formal no ambiente Maude, já que isto permite o uso de outras ferramentas formais atualmente disponíveis em especificações MSDF. Demonstramos

isto através de simulação e verificação de modelos de programas concorrentes e algoritmos distribuídos; (v) criando um exemplo de uma extensão não-trivial de Full Maude;)vi) finalmente, um dos objetos da ferramenta, ou seja, a integração de várias ferramentas formais usando Maude como tecnologia agregadora, foi recentemente testada através da extenção de MMT com a linguagem de estratégias para Maude de Verdejo [49] por Braga[1] e usando a ferramenta combinada para dar semântica ao cerne de GPH (Glasgow Parallel Haskell) [64].

[1]Comunicação pessoal, Fev. 2005.

APÊNDICE A – Constructive MSOS

Descrevemos neste apêndice o restante das construções CMSOS omitidas da seção 6.1.

A.1 Expressões

Começamos descrevendo os módulos para aplicação de operadores e de identificadores. A regra '`app-op1`' computa o argumento da aplicação—regras adicionais são necessários para os casos específicos. Como mencionamos anteriormente, a regra '`app-op2`' é uma conveniência para o usuário e é usada no caso específico em que o argumento original é uma tupla de valores e existe uma equação definida externamente que dá o resultado da aplicação da operação '`Op`' aos valores em '`Value*`'.

```
msos Cons/Exp/app-Op is
 Exp ::= app Op Arg .
 Value ::= tup Value* .

                          Arg -{...}-> Arg'
 [app-op1] -- ------------------------------------
           (app Op Arg) : Exp -{...}-> app Op Arg' .

                Value' := apply-op (Op, Value*)
 [app-op2] -- ------------------------------------
           (app Op (tup Value*)) : Exp --> Value' .
sosm
```

O módulo '`app-Id`' define a aplicação de um identificador a um argumento. As regras que governam cada caso específico serão definidas em módulos subseqüentes. Usualmente, um identificador será computado para um *operador* e a operação associada será aplicada.

```
msos Cons/Exp/app-Id is
 Exp ::= app Id Arg .

                 Id --> Id'
 -- --------------------------------
 (app Id Arg) : Exp --> app Id' Arg .

                Arg -{...}-> Arg'
 -- -------------------------------------
 (app Id Arg) : Exp -{...}-> app Id Arg' .
sosm
```

A.2 Declarações

O módulo '`simult-seq`' define a criação simultânea de um conjunto de amarrações. Isto significa que as amarrações são criadas independentes de umas das outras. Esta é uma variante da construção '`simult`' (não exibida aqui) que computa as amarrações em seqüência, ao passo que a versão mais genérica computa as amarrações sem uma ordem predefinida. As amarrações são criadas e então reunidas pelo operador '`+++`' ao final.

```
msos Cons/Dec/simult-seq is
 Dec ::= simult-seq Dec Dec .

                         Dec1 -{...}-> Dec'1
 -- ------------------------------------------------------
 (simult-seq Dec1 Dec2) : Dec -{...}-> simult-seq Dec'1 Dec2 .

                         Dec2 -{...}-> Dec'2
 -- ------------------------------------------------------
 (simult-seq Env1 Dec2) : Dec -{...}-> simult-seq Env1 Dec'2  .

        Env := (Env1 +++ Env2)
 -- ---------------------------------
 (simult-seq Env1 Env2) : Dec --> Env .
sosm
```

A construção '`accum`' define uma declaração acumulativa de amarrações onde as que forem definidas previamente estão disponíveis para as que estão sendo definidas em um determinado instante.

```
msos Cons/Dec/accum is
 Dec ::= accum Dec Dec .

 Label = {env : Env,...} .

                    Dec1 -{...}-> Dec'1
 -- ------------------------------------------------
 (accum Dec1 Dec2) : Dec -{...}-> accum Dec'1 Dec2 .

    Env' := (Env1 / Env0),
                    Dec2 -{env = Env', ...}-> Dec'2
 -- ------------------------------------------------------------
 (accum Env1 Dec2) : Dec -{env = Env0, ...}-> accum Env1 Dec'2 .

         Env := (Env2 / Env1)
 -- ------------------------------
 (accum Env1 Env2) : Dec --> Env .
sosm
```

A construção '`Exp/local`' recebeu uma declaração e uma expressão, e computa a expressão em termos da declaração. A declaração é inicialmente computada em '`Env`', um conjunto de amarrações que são usadas para sobrepor o ambiente atual, '`Env0`', que "encobre" a construção. Estas amarrações recém-criadas, '`Env'`', são usadas para computar cada passo da expressão '`Exp`'. Quando esta expressão atingir um valor final '`Value`', toda a construção é reescrita neste valor.

```
msos Cons/Exp/local is
 Exp ::= local Dec Exp .

 Label = { env : Env, ...} .

                   Dec -{...}-> Dec'
```

```
 -- ----------------------------------------
 (local Dec Exp) : Exp -{...}-> local Dec' Exp .

    Env' := (Env / Env0),
                   Exp -{env = Env', ...}-> Exp'
 -- ------------------------------------------------------
 (local Env Exp) : Exp -{env = Env0, ...}-> local Env Exp' .

 (local Env Value) : Exp --> Value .
sosm
```

A.3 Comandos

O módulo '`seq-Cmd-Exp`' define a construção '`seq`' que computa a seqüência de comandos dados como o seu primeiro argumento e então computa a expressão, produzindo seu valor. Pode ser vista como o corpo de uma função numa linguagem imperativa onde o último comando é um tipo de '`return`', que produz o valor de '`Exp`'.

```
msos Cons/Exp/seq-Cmd-Exp is
 Exp ::= seq Cmd Exp .

                 Cmd -{...}-> Cmd'
 -- ---------------------------------------
 (seq Cmd Exp) : Exp -{...}-> seq Cmd' Exp .

 (seq skip Exp) : Exp --> Exp .
sosm
```

A construção definida pelo módulo '`seq-Exp-Cmd`' é equivalente à definida em '`seq-Exp-Cmd`', mas a expressão é computada antes dos comandos, no entanto seu valor computado ainda assim é retornado após a execução dos comandos da operação.

```
msos Cons/Exp/seq-Exp-Cmd is
 Exp ::= seq Exp Cmd .

                 Exp -{...}-> Exp'
```

```
 -- ---------------------------------------
 (seq Exp Cmd) : Exp -{...}-> seq Exp' Cmd .

                   Cmd -{...}-> Cmd'
 -- -----------------------------------------
 (seq Value Cmd) : Exp -{...}-> seq Value Cmd' .

 (seq Value skip) : Exp --> Value .
sosm
```

A construção '`cond`' é um comando condicional, que seleciona qual comando será executado, dependendo da computação do seu primeiro argumento.

```
msos Cons/Cmd/cond is
 Cmd ::= cond Exp Cmd Cmd .
 Value ::= Boolean .

                        Exp -{...}-> Exp'
 -- ------------------------------------------------
 (cond Exp Cmd1 Cmd2) : Cmd -{...}-> cond Exp' Cmd1 Cmd2 .

 (cond tt Cmd1 Cmd2) : Cmd --> Cmd1 .

 (cond ff Cmd1 Cmd2) : Cmd --> Cmd2 .
sosm
```

A construção '`while`' é a construção *loop* tradicional, definida em termos de seqüência de comandos e um comando condicional.

```
msos Cons/Cmd/while is
 see Cons/Cmd/cond, Cons/Cmd/seq .

 Cmd ::= while Exp Cmd .

 (while Exp Cmd) : Cmd -->
   cond Exp (seq Cmd (while Exp Cmd)) skip .
sosm
```

As seguintes construções lidam com informação que pode ser alterada, modelada através de uma memória '`Store`', um componente escrita-leitura indexado por '`st`' e '`st'`'. Para atingir a generalidade desejada, CMSOS usa o conjunto '`Var`' para representar as variáveis que serão amarradas a locações de memória (representadas pelo conjunto '`Cell`').

```
msos Cons/Var is
 Var .
 Var ::= Cell .
sosm
```

A computação de uma variável como uma expressão espera que uma '`Cell`' seja obtida. Quando isto ocorrer, o valor amarrado à célula na memória é retornado.

```
msos Cons/Exp/Var is
 Exp ::= Var .
 Label = {store : Store, store' : Store, ...} .

    Var -{...}-> Var'
 -- ----------------------
 Var : Exp -{...}-> Var' .

               Value := lookup(Cell, Store)
 -- ------------------------------------------------
 Cell : Exp -{store = Store, store' = Store,-}-> Value .
sosm
```

Variáveis podem ou não ser relacionadas a identificadores. Quando forem, usamos o seguinte módulo, que define a computação de identificadores no contexto de variáveis. Espera-se que um identificador neste caso compute em uma célula. Para aparecer em ambientes, uma célula deve ser um valor '`Bindable`'.

```
msos Cons/Var/Id is
 Var ::= Id .
 Bindable ::= Cell .

 Label = {env : Env, ...} .
```

```
      Cell := lookup (Id, Env)
 -- ----------------------------
 Id : Var -{env = Env,-}-> Cell .
sosm
```

A construção '`assign-seq`' muda o valor apontado por '`Var`' para o valor obtido pela computação de '`Exp`'.

```
msos Cons/Exp/assign-seq is
 Exp ::= assign-seq Var Exp .

 Label = {store : Store, store' : Store, ...} .

                        Var -{...}-> Var'
 -- ------------------------------------------------------
 (assign-seq Var Exp) : Exp -{...}-> assign-seq Var' Exp .

                         Exp -{...}-> Exp'
 -- ------------------------------------------------------
 (assign-seq Cell Exp) : Exp -{...}-> assign-seq Cell Exp' .

 def lookup(Cell, Store),
              Store' :=  (Cell |-> Storable) / Store
 -- ---------------------------------------------
 (assign-seq Cell Storable) : Exp
   -{store  = Store, store' = Store',-}-> Storable .
sosm
```

Estas construção são complementares. A primeira '`ref`' computa a variável passada como seu argumento. A segunda '`deref`' computa a expressão e espera que seja computada numa '`ref Cell`'. Estas construções são usadas para trazer o aspecto imperativo a uma linguagem funcional através da criação de um "valor" especial '`ref`' que contém um ponteiro. Numa linguagem puramente imperativa estas construções não são necessárias, já que variáveis podem ser usadas diretamente. Elas funcionam da seguinte maneira: se quisermos amarrar um identificador `i` a um valor ν através de memória, primeiro criamos

uma nova célula `c` na memória que é amarrada a ν e usa esta célula como argumento da construção '`ref`'. O identificador então é amarrado indiretamente a ν. Para acessar o valor, chamamos '`deref i`', que irá ser computado em '`deref (ref (c))`'. A computação desta expressão, de acordo com as regras em '`Cons/Exp/Var`' será computada no valor ν.

```
msos Cons/Exp/ref is
 Exp ::= ref Var .
 Value ::= ref Cell .

             Var -{...}-> Var'
 -- ------------------------------
 (ref Var) : Exp -{...}-> ref Var' .
sosm

msos Cons/Var/deref is
 Var ::= deref Exp .
 Value ::= ref Cell .

               Exp -{...}-> Exp'
 -- ----------------------------------
 (deref Exp) : Var -{...}-> deref Exp' .

 (deref (ref Cell)) : Var --> Cell .
sosm
```

A seguinte construção retorna o valor amarrado à variável '`Var`'.

```
msos Cons/Exp/assigned is
 Exp ::= assigned Var .
 Label = {store : Store, store' : Store, ...} .
                  Var -{...}-> Var'
 -- ----------------------------------------
 (assigned Var) : Exp -{...}-> assigned Var' .

    Storable := lookup (Cell ,Store)
 -- ------------------------------------------------
```

```
 (assigned Cell) : Exp -{store = Store,
                        store' = Store,-}-> Storable .
sosm
```

A construção '`alloc`' cria uma nova entrada na memória para o valor obtido a partir da computação de '`Exp`' e retorna um ponteiro para esta entrada.

```
msos Cons/Var/alloc is
 Var ::= alloc Exp .

 Label = {store : Store, store' : Store, ...} .

               Exp -{...}-> Exp'
 -- ----------------------------------
 (alloc Exp) : Var -{...}-> alloc Exp' .

    Cell := new-cell (Store),
        Store' := (Cell |-> Storable) / Store
 -- ---------------------------------------
 (alloc Storable) : Var -{store = Store,
                          store' = Store',-}-> Cell .
sosm
```

A.4 Abstrações

Continuando a semântica de abstrações da seção 6.1,o módulo '`Cons/Dec/app`' define a sintaxe abstrata da aplicação de um argumento a um parâmetro. Esta aplicação será convertida numa declaração em que o argumento será amarrado ao parâmetro. Esta construção será usada na aplicação de *closures* a expressões.

```
msos Cons/Dec/app is
 Dec ::= app Par Arg .

                 Arg -{...}-> Arg'
 -- ---------------------------------------
 (app Par Arg) : Dec -{...}-> (app Par Arg') .
```

```
sosm
```

O módulo '`Cons/Par/bind`' define os parâmetros formais de abstrações e como eles se tornam declarações quando aplicados a argumentos.

```
msos Cons/Par/bind is
 see Cons/Dec/app .

 Par ::= bind Id .

 (app (bind Id) Bindable) : Dec --> (Id |-> Bindable) .
sosm
```

Para amarrar vários parâmetros simultaneamente, o seguinte módulo deve ser usado, o qual define a construção '`tup`'.

```
msos Cons/Par/tup is
 see Cons/Dec/app, Cons/Dec/simult .
 see Cons/Exp/tup .

 Par ::= tup Par* .

 (app (tup(Par, Par*)) (tup (Bindable, Bindable*))) : Dec -->
    (simult (app Par Bindable)
            (app (tup(Par*)) (tup(Bindable*)))) .

 (app tup() tup()) : Dec --> void .
sosm
```

Para amarrações recursivas, usamos o conceito de *finite unfolding*, com *reclosures*. Definimos uma amarração recursiva com a construção '`rec`' aplicada antes de uma declaração. Isto cria um tipo especial de declaração que sempre terá acesso a '`Dec`'. À medida que esta declaração é computada para gerar amarrações, '`rec Dec`' é computada novamente, tendo o efeito desejado de *finite unfolding*.

```
msos Cons/Dec/rec is
```

```
 see Cons/Dec/bind, Cons/Dec/simult,
     Cons/Dec/simult-seq, Cons/Exp/close,
     Cons/Abs/closure .

 Dec ::= rec Dec .
 Dec ::= reclose Dec Dec .

 (rec Dec) : Dec --> (reclose (rec Dec) Dec) .

 (reclose (rec Dec) (bind Id (close Abs))) : Dec -->
  (bind Id (close (closure (rec Dec) Abs))) .

 (reclose (rec Dec) (simult-seq Dec1 Dec2)) : Dec -->
 (simult-seq (reclose (rec Dec) Dec1) (reclose (rec Dec) Dec2)) .

 (reclose (rec Dec) (simult Dec1 Dec2)) : Dec -->
 (simult (reclose (rec Dec) Dec1) (reclose (rec Dec) Dec2)) .
sosm
```

A.5 Concorrência

A construção '`start`' sinaliza a criação de uma nova *thread*. Espera-se que o corpo da *thread* consista numa abstração que será aplicada à tupla vazia durante sua ativação. O módulo '`Cons/Cmd/start`' define seu significado: após a computação da expressão '`Exp`', a construção sinaliza a criação de uma nova *thread* que "produz" a abstração no componente apenas-escrita '`starting`'.

```
msos Cons/Cmd/start is
 Cmd ::= start Exp .

 Label = {starting' : Abs*, ...} .

 Value ::= Abs .

               Exp -{...}-> Exp'
 -- ------------------------------------
```

```
 (start Exp) : Cmd -{...}-> (start Exp') .

 (start Abs) : Cmd -{starting' = Abs,-}-> skip .
sosm
```

Sistemas inteiros são compostos de comandos. O seguinte módulo descreve a computação de um comando no contexto de um sistema. Se durante a execução de um comando, uma nova *thread* é detectada no componente '`starting'`', ela é removida do componente e colocada no conjunto de *threads* em execuação, reunidas através da construção '`conc`'. Se nenhuma nova *thread* é sinalizada, a execuação continua normalmente. À medida que *threads* terminam (computam para o comando '`skip`'), elas são removidas do conjunto.

```
msos Cons/Sys/Cmd is
 see Cons/Sys/conc, Cons/Cmd/effect,
     Cons/Exp/Abs,
     Cons/Exp/app, Cons/Exp/tup .

 Sys ::= Cmd | skip .

 Label = {starting' : Abs*, ...} .

      Cmd -{starting' = Abs, ...}-> Cmd'
-- ------------------------------------
Cmd : Sys -{starting' = (), ...}->
     conc Cmd' effect (app Abs tup()) .

     Cmd -{starting' = (), ...}-> Cmd'
 -- ------------------------------------
 Cmd : Sys -{starting' = (), ...}-> Cmd' .

 (conc skip Sys) : Sys --> Sys .

 (conc Sys skip) : Sys --> Sys .
sosm
```

Os módulos a seguir lidam com troca síncrona de mensagens. Vamos começar com a criação de canais, representados pelo conjunto '`Chan`'. Cada canal tem um identificador único.

```
msos Data/Chan is
 Chan .
 Chan ::= chan Int .
sosm
```

O '`alloc-chan`' cria um novo canal para ser usado e adiciona ao componente escrita-leitura '`Chans`', que gerencia todos os canais criados até o momento. A função '`new-chan`' é definida externamente através de equações para simplificar a especificação e cria um novo, não usado, canal.

```
msos Data/Chans is
 Chans .
 Chans = (Chan) Set .
sosm

msos Cons/Exp/alloc-chan is
 Exp ::= alloc-chan .

 Label = {chans : Chans, chans' : Chans, ...} .

 Value ::= Chan .
 Chan ::= new-chan (Chans) .

 Chan := new-chan (Chans), Chans' := Chans + { Chan }
 -- ------------------------------------------------------
 alloc-chan : Exp -{chans = Chans, chans' = Chans', -}-> Chan .
sosm
```

Para modelar a troca síncrona de mensagens, CMSOS segue a idéia de Concurrent ML. O componente de apenas-escrita '`event`' modela a produção de eventos durante uma computação. *Threads* ficam bloqueadas após a produção de eventos, esperando que outras *threads* produzam eventos que casem com os seus. Concurrent ML define diversos tipos

de eventos e suas relações de casamento, mas, neste caso, os únicos eventos são '`sending`' e '`receiving`', que modelam o envio e recebimento de mensagens através de um canal.

A função '`send-chan-seq`' recebe duas expressões como argumentos: a primeira é computada em um identificador de canal e a segunda é computada no valor a ser transmitido por aquele canal. Após estas computações, ela produz o evento '`sending`' com o canal e o valor.

```
msos Cons/Cmd/send-chan-seq is
 Cmd ::= send-chan-seq Exp Exp .

 Label = {event' : Event*, ...} .

 Event ::= sending Chan Value .

 Value ::= Chan .

              Exp1 -{...}-> Exp1'
 -- ----------------------------------------
 (send-chan-seq Exp1 Exp2) : Cmd -{...}->
                   (send-chan-seq Exp1' Exp2) .

              Exp2 -{...}-> Exp2'
 -- ----------------------------------------
 (send-chan-seq Chan Exp2) : Cmd -{...}->
                   (send-chan-seq Chan Exp2') .

 (send-chan-seq Chan Value) : Cmd
    -{event' = (sending Chan Value),-}-> skip .
sosm
```

A construção '`recv-chan`' recebe um valor através do canal que é obtido através da computação de seu argumento '`Exp`'. Aqui, desviamos da especificação original MSDF já que esta usou unificação de variáveis, uma característica não disponível em Maude 2.1.1. Originalmente, a função '`recv-chan`' também usava uma variável livre '`Value`' que se unifica quando há um casamento com um evento '`sending`'. Nossa soluçao é, após computar '`Exp`' em '`Chan`', colocarmos um *marcador* '`ph Chan`' no lugar da variável livre. Quando o

casamento ocorre, substituímos este marcador com o valor correto, usando a técnica que usamos para definir a Semântica de Reescrita Modular para Concurrent ML em [63]

```
msos Cons/Exp/recv-chan is
 Exp ::= recv-chan Exp .

 Label = {event' : Event*, ...} .

 Event ::= receiving Chan .

 Value ::= ph Chan | Chan .

                   Exp -{...}-> Exp'
 -- ------------------------------------------
 (recv-chan Exp) : Exp -{...}-> (recv-chan Exp') .

 (recv-chan Chan) : Exp -{event' = (receiving Chan),-}->
                                  (ph Chan) .
sosm
```

O módulo '`Cons/Sys/conc-chan`' descreve o casamento de eventos: se eventos '`sending`' e '`receiving`' forem produzidos por duas *threads*, eles sincronizam e o valor de um é passado para outro. Isto é feito usando uma *metafunção* '`update-ph`' que atualiza o marcador com o valor transmitido. Esta metafunção atua sobre o texto do programa no metanível para fazer a substituição, e para esta razão nossa solução não depende da assinatura da linguagem e por isso não afeta negativamente a modularidade da especificação.

```
msos Cons/Sys/conc-chan is
 Sys ::= conc Sys Sys .

 Sys ::= update-ph (Sys, Chan, Value) .

 Label = {event' : Event*, ...} .

 Event ::= sending Chan Value
```

```
         | receiving Chan .

 Sys1 -{event' = (sending Chan Value),-}-> Sys1',
        Sys2 -{event' = (receiving Chan),-}-> Sys2'
 -- ---------------------------------------------
 (conc Sys1 Sys2) : Sys -{event' = (),-}->
      (conc Sys1' update-ph (Sys2', Chan, Value)) .

 Sys2 -{event' = (sending Chan Value),-}-> Sys2',
         Sys1 -{event' = (receiving Chan),-}-> Sys1'
 -- ---------------------------------------------
 (conc Sys1 Sys2) : Sys -{event' = (),-}->
       (conc update-ph (Sys1', Chan, Value) Sys2') .
sosm
```

Finalmente, devemos evitar que *threads* continuem a executar se houver eventos que não foram casados. Isto é feito através da construção '`quiet`', que deixa o sistema evoluir sempre que todos os eventos estiverem casados. Quando isto ocorre, o componente '`event`' é sempre a seqüência vazia '`()`'.

```
msos Cons/Sys/quiet is
 see Cons/Sys/Cmd .

 Sys ::= quiet Sys .

 Label = {event' : Event*, ...} .

               Sys -{event' = (), ...}-> Sys'
 -- ----------------------------------------
 (quiet Sys) : Sys -{event' = (), ...}-> (quiet Sys') .

 (quiet skip) : Sys --> skip .
sosm
```

APÊNDICE B – Especificação da linguagem ML

Este capítulo descreve a especificação completa de ML, descrita na seção 6.1.2.

B.1 Expressões

Começamos descrevendo expressões ML a construções CMSOS equivalentes. Primeiro, precisamos reunir todos os módulos CMSOS necessários para as definições de expressões em ML. Isto é feito criando-se um módulo '`Lang/ML/Exp`'. O módulo contém referências explícitas para todas as construções CMSOS necessárias. Também define que o conjunto de valores ('`Value`') e operadores ('`Op`') são "amarráveis" em ambientes, que o conjunto de valores é "passável" para abstrações procedurais e que o conjunto de operadores contém as constantes '`plus`', '`times`', etc.

```
msos Lang/ML/Exp is
 see Cons/Prog, Cons/Prog/Exp .

 see Cons/Exp, Cons/Exp/Boolean, Cons/Exp/Int,
     Cons/Exp/Id, Cons/Exp/cond, Cons/Exp/app-Op,
     Cons/Exp/app-Id, Cons/Exp/tup, Cons/Exp/tup-seq .

 see Cons/Arg, Cons/Arg/Exp .

 see Cons/Op .

 see Cons/Id .

 Bindable ::= Value | Op .
```

```
 Op ::= plus | times | minus | eq | lt | gt .

 Passable ::= Value .
sosm
```

O seguinte módulo contém as definições básicas para expressões ML. Ele é definido como um módulo de sistema que inclui o módulo MSDF '`Lang/ML/Exp`'. Lembre-se que devemos definir a operação '`apply-op`' externamente e isto é feito aqui para cada constante '`Op`' declarada no módulo '`Lang/ML/Exp`'. Em seguida, criamos o ambiente inicial com as amarrações de identificadores para operadores. Reusamos os nomes dos operadores como identificadores, criados pela função de coerção '`ide`', no mapeamento.

```
mod Lang/ML/Exp' is
 including Lang/ML/Exp .
 including QID .

 vars i1 i2 : Int .

 eq apply-op (plus, (i1, i2)) = i1 + i2 .
 eq apply-op (minus, (i1, i2)) = i1 - i2 .
 eq apply-op (times, (i1, i2)) = i1 * i2 .
 eq apply-op (eq, (i1, i2)) = if i1 == i2 then tt else ff fi .
 eq apply-op (lt, (i1, i2)) = if i1 < i2 then tt else ff fi .
 eq apply-op (gt, (i1, i2)) = if i1 > i2 then tt else ff fi .

 op ide : Qid -> Id .
 op ide : Op -> Id .
 op op : Qid -> Op .

 eq init-env = (ide(eq) |-> eq +++ ide(lt) |-> lt +++
                ide(gt) |-> gt +++ ide(plus)  |-> plus +++
                ide(times) |-> times +++ ide(minus) |-> minus) .


 eq op ('+) = plus .   eq op ('*) = times .
 eq op ('-) = minus .  eq op ('<) = lt .
```

```
 eq op ('>) = gt .      eq op ('=) = eq .
endm
```

Começamos a gramática especificando "constantes especiais" (⟨ scon ⟩), que atualmente são inteiros apenas, representados pelo não terminal '`integer literal`'.

▶ *Constantes especiais*

⟨ scon ⟩ → ⟨ integer literal ⟩

Constantes especiais são passadas inalteradas para as construções CMSOS.

▶ *Infix operators*

Em seguida as regras para operadores "infixados" (⟨ inop ⟩). Podem ser ou o símbolo de "igual" ou algum outro símbolo definido pelo não-terminal ⟨ symbolic id ⟩. Espera-se que este não-terminal seja provido pela análise léxica. Ambos são convertidos para '`Op`' em CMSOS através da função de coerção '`op`' que recebe como argumento um qid.

⟨ inop ⟩ → '=' | ⟨ symbolic id ⟩

Seja s uma metavariável sobre ⟨ symbolic id ⟩.

⟦=⟧ = `op('=)`
⟦s⟧ = `op(s)`

▶ *Identificadores*

Identificadores definidos pelo não-terminal ⟨ id ⟩ são convertidos em '`Id`'s usando a função de coerção '`ide`' da mesma forma que a função '`op`'.

⟨ vid ⟩ → ⟨ id ⟩

Seja i uma metavariável sobre ⟨ vid ⟩.

$$[\![\mathtt{i}]\!] \;=\; \mathtt{ide(i)}$$

▶ *Expressões atômicas*

Dado que estamos lidando com uma versão abstrata da sintaxe de ML, iremos evitar o uso de diferentes não-terminais para representar expressões atômicas, de aplicação, infixadas e completas. Iniciamos abaixo com as regras para conversão de expressões atômicas, que são constantes especiais, identificadores e tuplas.

$$\langle\,\mathsf{exp}\,\rangle \rightarrow \langle\,\mathsf{scon}\,\rangle \mid \langle\,\mathsf{vid}\,\rangle \mid \text{`()'} \mid \text{`('}\ \langle\,\mathsf{exp}\,\rangle\ \text{`)'} \mid \text{`('}\ \langle\,\mathsf{exp}\,\rangle^*\ \text{`)'}$$

Sejam e_i metavariáveis sobre $\langle\,\mathsf{exp}\,\rangle$.

$$[\![\mathtt{()}]\!] \;=\; \mathtt{tup()}$$
$$[\![\mathtt{(}e\mathtt{)}]\!] \;=\; [\![e]\!]$$
$$[\![\mathtt{(}e*\mathtt{)}]\!] \;=\; \mathtt{tup\text{-}seq(}[\![e*]\!]\mathtt{)}$$

▶ *Expressões de aplicação*

Expressões de aplicação são usadas para chamar uma abstração procedural e são convertidas para a construção CMSOS 'app'.

$$\langle\,\mathsf{exp}\,\rangle \rightarrow \langle\,\mathsf{exp}\,\rangle\ \langle\,\mathsf{exp}\,\rangle$$

$$[\![e_0\ e_1]\!] \;=\; \mathtt{app}\ [\![e_0]\!]\ [\![e_1]\!]$$

▶ *Expressões infixadas*

Expressões infixadas contêm os operadores infixados, que são convertidos para a aplicação do operador que recebe como argumento a tupla seqüencial formada por ambas as expressões.

$$\langle\,\mathsf{exp}\,\rangle \rightarrow \langle\,\mathsf{exp}\,\rangle\ \langle\,\mathsf{inop}\,\rangle\ \langle\,\mathsf{exp}\,\rangle$$

Seja o uma metavariável sobre ⟨ inop ⟩ .

$$[\![e_0 \text{ o } e_1]\!] = \texttt{app } [\![\text{o}]\!] \texttt{ tup-seq}([\![e_0]\!],[\![e_1]\!])$$

▶ *Expressões completas*

Expressões completas são todas acima e as construções ML para condicionais.

⟨ exp ⟩ → ⟨ exp ⟩ 'andalso' ⟨ exp ⟩ | ⟨ exp ⟩ 'orelse' ⟨ exp ⟩
| 'if' ⟨ exp ⟩ 'then' ⟨ exp ⟩ 'else' ⟨ exp ⟩

$$[\![e_0 \texttt{ andalso } e_1]\!] = \texttt{cond } [\![e_0]\!] \; [\![e_1]\!] \texttt{ ff}$$
$$[\![e_0 \texttt{ orelse } e_1]\!] = \texttt{cond } [\![e_0]\!] \texttt{ tt } [\![e_1]\!]$$
$$[\![\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2]\!] = \texttt{cond } [\![e_0]\!] \; [\![e_1]\!] \; [\![e_2]\!]$$

B.2 Declarações

Para declarações, vamos introduzir o módulo MSDF relevante que contém todas as construções CMSOS necessárias.

```
msos Lang/ML/Dec is
 see Lang/ML/Exp' .

 see Cons/Prog, Cons/Prog/Dec .

 see Cons/Dec, Cons/Dec/bind, Cons/Dec/simult-seq,
     Cons/Dec/accum, Cons/Dec/local .

 see Cons/Exp, Cons/Exp/local .
sosm
```

▶ *Expressões "let"*

Começamos estendendo as expressões atômicas com a expressão '`let-in-end`', que é mapeada na construção CMSOS '`local`'.

$\langle \mathsf{exp} \rangle \rightarrow$ 'let' $\langle \mathsf{dec} \rangle$ 'in' $\langle \mathsf{exp} \rangle$ 'end'

Seja $\mathfrak{d}$ uma metavariável sobre $\langle \mathsf{dec} \rangle$.

$[\![\texttt{let d in } e \texttt{ end}]\!] = \texttt{local } [\![\mathfrak{d}]\!] \; [\![e]\!]$

▶ *Amarrações de valores*

Em seguida, as declarações são definidas. Amarrações de valores são convertidas na construção CMSOS '`bind`'.

$\langle \mathsf{dec} \rangle \rightarrow$ 'val' $\langle \mathsf{vid} \rangle$ '=' $\langle \mathsf{exp} \rangle$

$[\![\texttt{val i = } e]\!] = \texttt{bind } [\![\mathfrak{i}]\!] \; [\![e]\!]$

B.3 Imperativos

ML não tem o conceito de um "comando," já que tudo é uma expressão nesta linguagem, mas usa aspectos imperativos. O módulo '`Lang/ML/Cmd`' define os "comandos" na linguagem ML:

```
msos Lang/ML/Cmd is
 see Lang/ML/Dec .

 see Cons/Cmd, Cons/Cmd/seq-n, Cons/Cmd/effect, Cons/Cmd/while .

 see Cons/Exp, Cons/Exp/seq-Cmd-Exp, Cons/Exp/seq-Exp-Cmd,
     Cons/Exp/assign-seq, Cons/Exp/ref, Cons/Exp/assigned .

 see Cons/Var, Cons/Var/alloc, Cons/Var/deref .

 Storable ::= Value .
sosm
```

Em seguida adicionamos uma outra definição "externa", que é a equação que aloca uma nova célula numa memória.

```
mod Lang/ML/Cmd' is
 including Lang/ML/Cmd .

 var Store : Store .
 eq new-cell (Store) = cell (length (Store) + 1) .
endm
```

▶ *Seqüenciamento de expressões*

Começamos por rever as expressões atômicas, onde definimos a sintaxe do seqüênciamento de expressões. Seqüências de expressões são convertidas na construção '`seq-Cmd-Exp`' que recebe uma seqüência de comandos e uma expressão. Cada comando é uma expressão envolvida pela construção '`effect`'. Por exemplo, a seqüência de expressões '`3;4;1`' é convertida em '`seq seq (effect (3), effect (4)), 1`'.

$\langle$ exp $\rangle \rightarrow$ '(' $\langle$ exp seq $\rangle$ ';' $\langle$ exp $\rangle$ ')'
$\langle$ exp seq $\rangle \rightarrow \langle$ exp $\rangle$ ';' $\langle$ exp seq $\rangle$

Seja *es* uma metavaríavel sobre $\langle$ exp seq $\rangle$.

$[\![$ (*es* ; *e*) $]\!]$ = `seq (seq` $[\![es]\!]$) $[\![e]\!]$
$[\![e$; *es* $]\!]$ = `(effect` $[\![e]\!]$,$[\![es]\!]$`)`

▶ *Resolução de ponteiros*

Para resolver uma expressão-ponteiro, primeiro computamos a expressão em uma célula com a construção '`deref`' e produzimos o valor atribuído a esta célula com a construção '`assigned`'.

$\langle$ exp $\rangle \rightarrow$ '!' $\langle$ exp $\rangle$

$[\![$! *e* $]\!]$ = `assigned (deref (`$[\![e]\!]$`))`

▶ *Criação de ponteiros*

Inicialmente, alocamos uma nova célula na memória e construímos um valor com '`ref`'

$$\langle \mathsf{exp} \rangle \rightarrow \text{`}\texttt{ref}\text{'} \ \langle \mathsf{exp} \rangle$$

$$[\![\texttt{ref } e]\!] \ = \texttt{ref (alloc } ([\![e]\!]))$$

▶ *Atribuição*

Já que uma atribuição em ML não contém um valor final, usamos a construção '`seq-Cmd-Exp`' para primeiro executar a atribuição e depois para retornar a tupla vazia ('`tup()`'). A atribuição propriamente dita é feita de-referenciando a expressão atribuída e em seguida usando a construção '`assign-seq`'.

$$\langle \mathsf{exp} \rangle \rightarrow \langle \mathsf{exp} \rangle \ \text{`}\texttt{:=}\text{'} \ \langle \mathsf{exp} \rangle$$

$$[\![e_0 \texttt{ := } e_1]\!] \ = \texttt{seq (effect (assign-seq (deref } [\![e_0]\!]\texttt{) } [\![e_1]\!]\texttt{)) tup()}$$

▶ *Loops*

Finalmente, adicionamos uma construção típica de uma linguagem imperativa, o comando de *loop*.

$$\langle \mathsf{exp} \rangle \rightarrow \langle \mathsf{while} \rangle \ \langle \mathsf{exp} \rangle \ \text{`}\texttt{do}\text{'} \ \langle \mathsf{exp} \rangle$$

$$[\![\texttt{while } e_0 \texttt{ do } e_1]\!] \ = \texttt{seq (while } [\![e_0]\!] \texttt{ (effect } [\![e_1]\!]\texttt{)) tup()}$$

B.4 Abstrações

O módulo '`Lang/ML/Abs`' é necessário para a definição de abstrações.

```
msos Lang/ML/Abs is
 see Lang/ML/Dec .

 see Cons/Exp, Cons/Exp/Abs, Cons/Exp/close, Cons/Exp/app-seq .
 see Cons/Abs, Cons/Abs/abs-Exp, Cons/Abs/closure .
 see Cons/Par, Cons/Par/bind, Cons/Par/tup .
 see Cons/Dec, Cons/Dec/app, Cons/Dec/rec .
sosm
```

▶ *Funções recursivas*

A versão de funções recursivas aqui é bastante simples e não faz uso de casamento de padrões mais complexos, já que optamos por demonstrar esta funcionalidade na semântica da linguagem Mini-Freja, apêndice D. A nova opção para o não-terminal ⟨ dec ⟩ mostra a sintaxe de funções recursivas: o primeiro ⟨ vid ⟩ é o nome da função, o segundo é o (único) parâmetro formal e o ⟨ exp ⟩ é o corpo. Ele é convertido para a amarração do nome da função a uma *closure*.

⟨ dec ⟩ → 'fun' ⟨ vid ⟩ ⟨ vid ⟩ '=' ⟨ exp ⟩

$[\![\texttt{fun } i_0\ i_1 \texttt{ = } e]\!] \texttt{ = rec (bind } [\![i_0]\!] \texttt{ (close (abs (bind } [\![i_1]\!]\texttt{) } [\![e]\!]\texttt{)))}$

B.5 Concorrência

Finalmente, vamos mostrar as primitivas de concorrência em ML. Esta seção mostra as primitivas para a criação de novas *threads* e para programas ML completos.

O módulo 'Lang/ML/Conc' reúne os módulos necessários.

```
msos Lang/ML/Conc is
 see Lang/ML/Cmd', Lang/ML/Abs .

 see Cons/Cmd, Cons/Cmd/send-chan-seq,
     Cons/Cmd/start .
```

```
 see Cons/Exp, Cons/Exp/recv-chan,
     Cons/Exp/alloc-chan .

 see Cons/Sys, Cons/Sys/Cmd, Cons/Sys/conc,
     Cons/Sys/conc-chan, Cons/Sys/quiet .
sosm
```

▶ *Criação de novas* threads

A operação '`spawn`' cria um novo processo, e é equivalente à construção CMSOS '`start`'.

⟨ exp ⟩ → '`spawn`' ⟨ exp ⟩

⟦`spawn` *e*⟧ = `seq (start` ⟦*e*⟧`)`

▶ *Criação de canais*

A declaração '`chan`' cria um novo canal, amarra-o ao identificador passado como seu argumento.

⟨ dec ⟩ → '`chan`' ⟨ vid ⟩

⟦`chan i`⟧ = `bind` ⟦i⟧ `alloc-chan`

▶ *Enviando e recebendo valores*

As operações '`send`' e '`receive`' transmitem informações através de um canal e são implementadas, respectivamente, pelas construções CMSOS '`send-chan-seq`' e '`recv-chan`'.

⟨ exp ⟩ → '`send`' '`(`' ⟨ exp ⟩ '`,`' ⟨ exp ⟩ '`)`' | '`receive`' ⟨ exp ⟩

⟦`send (` e_0 `,` e_1 `)`⟧ = `seq (send-chan-seq` ⟦e_0⟧ ⟦e_1⟧`) tup()`
⟦`receive` *e*⟧ = `recv-chan` ⟦*e*⟧

▶ *Programas concorrentes completos*

Todos os programas concorrentes ML devem ser prefixados por '`cml`', que é convertido para a construção CMSOS '`quiet`'.

$$\langle\, \texttt{exp} \,\rangle \rightarrow \text{`}\texttt{cml}\text{'} \langle\, \texttt{exp} \,\rangle$$

$$[\![\texttt{cml}\ e]\!] = \texttt{quiet (effect}\ [\![e]\!]\texttt{)}$$

APÊNDICE C – Especificação da linguagem MiniJava

Este capítulo contém a especificação completa de MiniJava, descrita inicialmente na seção 6.1.3.

C.1 Expressões

Expressões em MiniJava consistem de operações matemáticas, identificadores, chamadas de método (que sempre retornam um valor), constantes numéricas e objetos.

⟨ exp ⟩ → ⟨ math operation ⟩ | ⟨ id ⟩ | ⟨ method invocation ⟩
| ⟨ literal ⟩ | ⟨ this ⟩ | ⟨ new ⟩

► *Operações matemáticas*

⟨ math operation ⟩ → ⟨ exp ⟩ ⟨ math op ⟩ ⟨ exp ⟩
⟨ math op ⟩ → '&&' | '<' | '+' | '/' | '%' | '-' | '*' | '>' | '<=' | '>=' | '='

Sejam e_i metavariáveis sobre ⟨ exp ⟩, e m sobre ⟨ math op ⟩.

$[\![e_0 \;\; m \;\; e_1]\!] = \texttt{app}\ [\![m]\!]\ \texttt{tup-seq}\ ([\![e_0]\!], [\![e_1]\!])$

► *Identificadores*

Seja i uma metavariável sobre ⟨ id ⟩. Quando i não é o lado esquerdo de uma atribuição.

$[\![\mathtt{i}]\!] \;=\; \mathtt{assigned\ (deref\ i)}$

Caso contrário:

$[\![\mathtt{i}]\!] \;=\; \mathtt{deref\ i}$

▶ *Chamadas de métodos*

⟨ method invocation ⟩ → ⟨ exp ⟩ '.' ⟨ id ⟩ '(' ⟨ exp ⟩*')'

$[\![\mathtt{e\ .\ i\ (\ e*\ )}]\!] \;=\; \mathtt{app\ (app\ nth(n)\ [\![e]\!])\ (tup\text{-}seq\ p)}$

A computação de e deve retornar um objeto; n(i) é o número do método na classe do objeto retornado por e, obtido buscando o nome do método i na meta-informação sobre a classe gerada durante a fase de análise estática; e p é construído como uma seqüência de '`ref (alloc` $[\![e_i]\!]$`)`' que aloca uma nova entrada na memória para cada parêmtro e_i e $e*$.

▶ *Literais*

⟨ literal ⟩ → ⟨ boolean literal ⟩ | ⟨ integer literal ⟩
⟨ boolean literal ⟩ → '`true`' | '`false`'

Literais são convertidos diretamente para CMSOS.

▶ *Auto-referência*

⟨ this ⟩ → '`this`'

$[\![\mathtt{this}]\!] \;=\; \mathtt{app\ self\ tup()}$

▶ *Instanciação de objetos.*

⟨ new ⟩ → '`new`' ⟨ id ⟩ '`()`'

$[\![\mathtt{new\ i\ ()}]\!] \;=\; \mathtt{app\ i\ tup()}$

C.2 Comandos

MiniJava contém os comandos usuais encontrados em linguagens imperativas: condicionais, loops, saída, atribuição de valores, etc.

$$\langle \mathsf{statement} \rangle \rightarrow \langle \mathsf{if} \rangle \mid \langle \mathsf{while} \rangle \mid \langle \mathsf{block} \rangle \mid \langle \mathsf{print} \rangle \mid \langle \mathsf{assign} \rangle \mid \langle \mathsf{empty} \rangle$$

▶ *Condicionais*

$$\langle \mathsf{if} \rangle \rightarrow \text{`}\texttt{if}\text{'} \ \langle \mathsf{exp} \rangle \ \text{`}\texttt{then}\text{'} \ \langle \mathsf{statement} \rangle \ \text{`}\texttt{else}\text{'} \ \langle \mathsf{statement} \rangle$$

Sejam s_i metavariáveis sobre $\langle \mathsf{statement} \rangle$.

$$[\![\texttt{if } e \texttt{ then } \mathsf{s}_0 \texttt{ else } \mathsf{s}_1]\!] = \texttt{cond } [\![e]\!] \ [\![\mathsf{s}_0]\!] \ [\![\mathsf{s}_1]\!]$$

▶ *Loops*

$$\langle \mathsf{while} \rangle \rightarrow \text{`}\texttt{while}\text{'} \ \text{`}\texttt{(}\text{'} \ \langle \mathsf{exp} \rangle \ \text{`}\texttt{)}\text{'} \ \langle \mathsf{statement} \rangle$$

$$[\![\texttt{while (} e \texttt{) } s]\!] = \texttt{while } [\![e]\!] \ [\![s]\!]$$

▶ *Comandos em bloco*

$$\langle \mathsf{block} \rangle \rightarrow \text{`}\{\text{'} \ (\langle \mathsf{statement} \rangle)^* \text{`}\}\text{'}$$

$$[\![\{ \ \mathsf{s}* \ \}]\!] = \texttt{seq } [\![\mathsf{s}*]\!]$$

▶ *Saída*

$$\langle \mathsf{print} \rangle \rightarrow \text{`}\texttt{System.out.println}\text{'} \ \text{`}\texttt{(}\text{'} \ \langle \mathsf{exp} \rangle \ \text{`}\texttt{)}\text{'}$$

$[\![$`System.out.println ( e )`$]\!]$ = `print` $[\![e]\!]$

▶ *Atribuição*

⟨ assign ⟩ → ⟨ exp ⟩ '=' ⟨ exp ⟩

$[\![e_0$ `=` $e_1]\!]$ = `effect (assign-seq` $[\![e_0]\!]$ $[\![e_1]\!]$`)`

▶ *Comando inócuo*

⟨ empty ⟩ → ';'

$[\![;]\!]$ = `skip`

C.3 Classes

▶ *Declaração de classes*

Como descrevemos no início desta seção, uma declaração de classe define um objeto "protótipo", que é uma *closure* cujos campos tornam-se amarrações e os métodos tornam-se projeções de uma tupla.

⟨ class declaration ⟩ → '`class`' ⟨ identifier ⟩ '{' (⟨ field declaration ⟩)*
(⟨ method declaration ⟩)*'}'

Sejam $\mathtt{f}_i$ metavariáveis sobre ⟨ field declaration ⟩ e $\mathtt{m}_i$ sobre ⟨ method declaration ⟩.

$[\![$`class i { f` $*$ `m` $*$ `}`$]\!]$ =

```
   local
    (accum (accum [[f*]])
           (rec (bind self
                  close (abs (bind dummy) tup-seq ([[m]])))))
    app self tup()
```

▶ *Declaração da classe principal*

⟨ main class declaration ⟩ → 'class' ⟨ identifier ⟩
'{' ⟨ main method declaration ⟩ '}'

```
⟦class i { m }⟧ =
   local
    (accum void
           (rec (bind self
                  close (abs (bind dummy) tup-seq (⟦m⟧)))))
    app self tup()
```

A principal diferença é a ausência de declarações de campos e a existência de um único método.

▶ *Declarações de campos*

As informações de tipo são usadas apenas na fase de análise estática. Dado que o único tipo primitivo é o inteiro, amarramos o identificador `i` para uma célula recém-alocada, com o valor inicial zero.

⟨ field declaration ⟩ → ⟨ type ⟩ ⟨ identifier ⟩

Seja `t` uma metavariável sobre ⟨ type ⟩.

```
⟦t i⟧ = bind i (ref (alloc 0))
```

▶ *Declarações de métodos*

Uma declaração de métodos é convertida numa *closure* cuja lista de parâmetros formais é declarada através da construção '`tup`'. O corpo da *closure* é uma definição '`local`' com as declarações de variáveis na declaração e o corpo do método como a expressão que está sendo computada. Usamos o comando '`seq-Cmd-Exp`' de forma que a última expressão a ser computada é o valor de retorno do método.

⟨ method declaration ⟩ → ⟨ type ⟩ ⟨ identifier ⟩ '(' (⟨ parameter ⟩)*')'
'{' (⟨ var declaration ⟩)*(⟨ statement ⟩)*'return' ⟨ expression ⟩ '}'

Seja $\mathtt{p_i}$ uma metavariável sobre ⟨ parameter ⟩.

$$[\![\mathtt{t\ i\ (\ p*\ )\ \{\ v*\ \ s*\ return\ e\ \}}]\!] =$$
$$\mathtt{close\ (abs\ tup([\![p*]\!])\ (local\ (accum\ [\![v*]\!])\ (seq\ (seq\ [\![s*]\!])\ e)))}$$

▶ *Declaração do método 'main'*

⟨ main method declaration ⟩ → 'public static void main (String arg[]) {'
(⟨ statement ⟩)*'}'

$$[\![\mathtt{public\ static\ void\ main\ (String\ arg[])\ \{\ s*\ \}}]\!] =$$
$$\mathtt{close\ (abs\ tup(@)\ (local\ void\ (seq\ (seq\ [\![s*]\!])\ 0)))}$$

O método 'main' é similar a um método genérico, com a exceção de que não recebe argumentos, declarações de variáveis, nem tem nenhuma expressão de retorno. Neste caso, ele retorna o valor zero.

▶ *Programas completos*

Definimos agora como o corpo principal do programa convertido é criado. Ele consiste em uma série de amarrações dos nomes das classes para os objetos-protótipo obtidos a partir das declarações de classes. O corpo é uma chamada à primeira projeção da classe, que é exatamente o método 'main'.

Seja $\mathtt{cd_i}$ uma classe que não a principal e *exec* o corpo do programa principal.

$$[\![\mathtt{goal}]\!] = \mathtt{local\ (accum\ [\![cd*]\!])\ [\![}\mathit{exec}\mathtt{]\!]}$$

A conversão das declarações de classes é uma série de amarrações de nomes de classe ($\mathtt{n_i}$) para os objetos-protótipo ($\mathtt{o_i}$). Assim, para uma declaração de classe $\mathtt{i}$, $[\mathtt{cd_i}]$ é:

$$[\![\mathtt{cd_i}]\!] = \mathtt{bind\ n\ o_i}$$

Agora a equação para exec. É uma chamada ao método principal da classe principal. Apesar deste método não receber argumentos, uma tupla vazia com uma única referência é passada como um parâmetro inócuo. O objeto protótipo da classe principal é representado por o_m.

```
⟦exec⟧ =
 (app (app nth(0) o_m)
      (tup-seq (ref (alloc 0))))
```

O módulo abaixo reúne todos os módulos CMSOS necessários para a semântica.

```
msos MiniJava is
 see Cons/Prog, Cons/Prog/Exp .
 see Cons/Exp, Cons/Exp/Boolean, Cons/Exp/Int, Cons/Exp/local,
     Cons/Exp/Id, Cons/Exp/cond, Cons/Exp/app-Op,
     Cons/Exp/app-Id, Cons/Exp/tup, Cons/Exp/tup-seq .
 see Cons/Arg, Cons/Arg/Exp .
 see Cons/Op .
 see Cons/Id .
 see Cons/Prog, Cons/Prog/Dec .
 see Cons/Dec, Cons/Dec/bind, Cons/Dec/simult-seq,
     Cons/Dec/accum, Cons/Dec/local .
 see Cons/Exp, Cons/Exp/local .
 see Cons/Cmd, Cons/Cmd/seq-n, Cons/Cmd/effect, Cons/Cmd/while,
     Cons/Cmd/print .
 see Cons/Exp, Cons/Exp/seq-Cmd-Exp, Cons/Exp/seq-Exp-Cmd,
     Cons/Exp/assign-seq, Cons/Exp/ref, Cons/Exp/assigned .
 see Cons/Var, Cons/Var/alloc, Cons/Var/deref .
 see Cons/Exp/ref .
 see Cons/Exp, Cons/Exp/Abs, Cons/Exp/close, Cons/Exp/app-seq .
 see Cons/Abs, Cons/Abs/abs-Exp, Cons/Abs/closure .
 see Cons/Par, Cons/Par/bind, Cons/Par/tup .
 see Cons/Dec, Cons/Dec/app, Cons/Dec/rec .

 Bindable ::= Value | Op .
 Op ::= nth (Int) | plus | times | minus | eq | lt | gt .
```

```
 Passable ::= Value .
 Storable ::= Value .
sosm
```

O seguinte código define o ambiente inicial para programas MiniJava, com o registro inicial e as equações para '`apply-op`' e '`new-cell`'.

```
mod MiniJava' is
 including MiniJava .

 var I : Int .
 var V : Value .
 var VL : Seq`(Value`) .

 op init-rec : -> Record .
 eq init-rec = { out' = (()).Seq`(Value`),
                 env = (void).Map`(Id`|`Bindable`),
                 store = (void).Map`(Cell`|`Storable`) } .

 vars i1 i2 : Int .
 eq apply-op (plus, (i1, i2)) = i1 + i2 .
 eq apply-op (minus, (i1, i2)) = i1 - i2 .
 eq apply-op (times, (i1, i2)) = i1 * i2 .
 eq apply-op (eq, (i1, i2)) = if i1 == i2 then tt else ff fi .
 eq apply-op (lt, (i1, i2)) = if i1 < i2 then tt else ff fi .
 eq apply-op (gt, (i1, i2)) = if i1 > i2 then tt else ff fi .
 eq apply-op (nth (0),  (V, VL)) = V .
 ceq apply-op (nth (I),  (V, VL))
 = apply-op (nth (I - 1),  (VL))
 if I > 0 .

 op ide : Qid -> Id .
 op ide : Op -> Id .

 eq init-env = (ide(eq) |-> eq +++ ide(lt) |-> lt +++
                ide(gt) |-> gt +++ ide(plus)  |-> plus +++
```

```
                ide(times) |-> times +++ ide(minus) |-> minus) .

 var Store : Store .
 eq new-cell (Store) = cell (length (Store) + 1) .

 sort Output .
 op output : Conf -> Output .
 op output : Seq‘(Value‘) -> Output .

 rl output(< V:Value ::: ’Exp, { out’ = VL:Seq‘(Value‘),
    PR:PreRecord } >) => output(VL:Seq‘(Value‘)) .

endm)
```

APÊNDICE D – Especificação da linguagem Mini-Freja

Este capítulo complementa a semântica da linguagem MiniFreja, apresentada na seção 6.2, com a adição de regras para casamento de padrões.

O casamento de padrões é feito pela construção '`case Exp of Rules`', onde '`Rules`' é uma seqüência de opções a ser casada, com uma expressão resultante.

```
Rules .
Rule .

Rule ::= Pat => Exp .

Rules ::= Rule
        | Rules || Rules [assoc] .
```

Cada regra é um padrão a ser casado junto com a expressão resultante do casamento. Padrões seguem a mesma sintaxe de expressões: podemos casar com variáveis, contantes e listas.

```
Pat .
Pat ::= Pat :: Pat [assoc] .
Pat ::= p Const | p Var .
```

Infelizmente, devido a problemas de preregularidade, precisamos da função de coerção '`p_`' que converte constantes e variáveis para o conjunto de padrões. O problema ocorre com o operador de construção de listas '`Pat :: Pat`': se não tivéssemos usado a função de coerção, um termo como '`3 :: 5`' não teria um *sort mínimo*, já que poderia ser ou uma '`Pat`' ou uma '`Exp`'.

As seguintes regras implementam o casador de padrões para a linguagem Mini-Freja. Começamos criando uma operação adicional '`case(ν,R)`', que casa um valor ν (obtido através de uma expressão) com um conjunto R de regras.

```
Value ::= case (Value, Rules) .

       Exp ==> Value,
             case (Value, Rules) ==> Value'
[case] -- --------------------------------
       case Exp of Rules : Exp ==> Value' .
```

Casamentos são definidos de acordo com a seguinte assinatura. Quando um casamento é bem sucedido, ele computa em '`myes(ρ)`', onde ρ é a amarração resultante do casamento. Caso contrário, ele computa em '`mno`'.

```
Match .
Match ::= myes (Env) | mno .
```

Primeiro, o caso base, onde o conjunto de regras R consiste em uma única regra. A função '`match(ν,p)`' casa ν contra o padrão p e retorna ou '`myes(ρ)`', se for bem sucedida, ou '`mno`'. Se o valor '`Value`' casa com o padrão '`Pat`', então a expressão '`Exp`' é computada sobrescrevendo o ambiente atual com ρ.

```
Match ::= match (Value, Pat) .

       match (Value, Pat) ={env = Env, -}=> myes(Env'),
       Env'' := Env' / Env, Exp ={env = Env'', -}=> Value'
[case1] -- ------------------------------------------------
       case (Value, Pat => Exp) : Value
                                ={env = Env, -}=> Value' .
```

Vamos agora ao caso genérico, em que existem ao menos duas regras em R. A regra abaixo especificia o seguinte: o resultado do casamento de '`Value`' com '`Pat`' é dada pela função '`case-choose`' function, que retorna um valor '`Value''`

```
       match (Value, Pat) ==> Match,
```

```
       case-choose (Match, Exp, Value, Rules) ==> Value'
[case2] -- ------------------------------------------
       case (Value, ((Pat => Exp) || Rules)) : Value
                                          ==> Value' .
```

Abaixo a definição da função auxiliar '`case-choose(`$\mathtt{m}$`,`e`,`ν`,R)`'. Se o casamento $\mathtt{m}$ é '`mno`', significa que o casamento contra a primeira regra de R não foi bem sucedido; então o valor deve ser casado contra o restante das regras.

```
Value ::= case-choose (Match, Exp, Value, Rules) .

              case (Value, Rules) ==> Value'
[case-choose] -- ---------------------------------------
              case-choose (mno, Exp, Value, Rules) : Value
                                          ==> Value' .
```

Caso contrário, se o casamento for bem-sucedido, '`case-choose`' funciona de maneira similar à regra '`[case1]`'.

```
              Env'' := Env' / Env,
              Exp ={env = Env'', -}=> Value'
[case-choose] -- ------------------------------------------
              case-choose (myes(Env'),Exp,Value,Rules) : Value
                                  ={env = Env, -}=> Value' .
```

As seguintes regras especificam cada tipo possível de casamento de padrões. Primeiro, um casamento de um valor contra um padrão que é uma variável é sempre bem sucedido e cria uma amarração entre a variável e o valor.

```
[match-var] match (Value, p Var) : Match
            ={env = Env, -}=> myes (Var |-> Value / Env) .
```

Casando uma constante contra uma constante, será bem sucedido se e somente se ambas as constantes forem iguais.

```
                    Match := if Const1 == Const2
```

```
                              then myes (Env)
                              else mno fi
[match-const-const] -- ------------------------------
                    match (Const1, p Const2) : Match
                              ={env = Env, -}=> Match .
```

O casamento de '`cons`' com uma lista necessita de uma função auxiliar '`match-pair(m,v,p)`' que irá iterar sobre a lista, reunindo as amarrações, se os casamentos forem bem-sucedidos.

```
Match ::= match-pair (Match, Value, Pat)
```

A regra '`[match-cons-cons]`' diz que, quando casando uma lista contra uma outra lista, primeiro tentamos casar os primeiros elementos, e em seguida casamos o restante com '`match-pair`'.

```
                    match (Value1, Pat1) ==> Match1,
                    match-pair (Match1, Value2, Pat2) ==> Match2
[match-cons-cons] -- ----------------------------------------
                    match (cons (Value1, Value2),
                           Pat1 :: Pat2) : Match ==> Match2 .
```

Na regra '`[match-cons-cons]`', '`match-pair`' recebe o casamento do primeiro elemento de ambas as listas ('`Match1`'). Se este casamento não for bem sucedido, então o casamento das duas listas também não será.

```
[match-pair] match-pair (mno, Value, Pat) : Match ==> mno .
```

Caso contrário, ele casa recursivamente com o restante da lista, reunindo as amarrações produzidas.

```
               Env'' := Env' / Env,
               match (Value, Pat) ={env = Env'', -}=> Match
[match-pair] -- ------------------------------------------
               match-pair (myes (Env'), Value, Pat) : Match
                                  ={env = Env, -}=> Match .
```

Finalmente, os seguintes casamentos devem sempre ser mal sucedidos.

```
[match-const-cons] match (Const, Pat1 :: Pat2) : Match
                   ==> mno .
[match-cons-const] match (cons(Value1,Value2),p Const) : Match
                   ==> mno .
```

APÊNDICE E – Algoritmos distribuídos

Este apêndice completa o capítulo 6.3 com mais exemplos de algoritmos distribuídos. A seção E.1 descreve a especificação de um algoritmo de exclusão mútua usando semáforos; a seção E.2 continua a especificação dos Filósofos Glutões da seção 6.3.2.2 com algumas verificações adicionais; a seção E.3 mostra o algoritmo da Padaria de Lamport para exclusão mútua, cuja verificação fez uso de uma abstração equacional [65, 66]; finalmente, a seção E.4 mostra mais exemplos de model checking com um algoritmo para a eleição de líder num anel assíncrono.

E.1 Exclusão mútua com semáforos

Esta seção especifica um algoritmo de exclusão mútua usando semáforos e também serve como um exmplo introdutório de como um processo mantém o seu estado interno através do conjunto '`St`'.

Vamos começar com uma especificação sem semáforos e verificar a condição de corrida que acontecerá. Nesta especificação, processos têm dois estados possíveis: ou estão dentro de sua região crítica ('`cric`') ou não ('`rem`').

```
St .
St ::= crit | rem .

Proc .
Proc ::= pid (Int, St) .
```

Processos entram e saem de suas regiões críticas continuamente.

```
prc (Int, rem) : Proc --> prc (Int, crit) .
prc (Int, crit) : Proc --> prc (Int, rem) .
```

Como a especificação é bem simples, podemos buscar por todos os possíveis estados do sistema. Uma busca simples é suficiente para mostrar todas as quatro opções.

```
(search
  (< (prc (0, rem) prc (1, rem)) ::: 'Soup,
     { null } >) =>* C:Conf .)

Solution 1
C:Conf <- <(prc(0,rem) prc(1,rem))::: 'Soup,{null}>

Solution 2
C:Conf <- <(prc(0,crit) prc(1,rem))::: 'Soup,{null}>

Solution 3
C:Conf <- <(prc(0,rem) prc(1,crit))::: 'Soup,{null}>

Solution 4
C:Conf <- <(prc(0,crit) prc(1,crit))::: 'Soup,{null}>
```

Para evitar a condição de corrida exemplificada na quarta solução, iremos reescrever a especificação usando um semáforo: antes de entrar na região crítica, um processo passará pelos estados intermediários '`down`' e '`up`'.

```
Label = { sem : Int, sem' : Int, ... } .
```

Os estados '`down`' e '`up`' fazem parte de '`St`':

```
St ::= down | up .
```

Antes de entrar em sua região crítica, um processo entra no seu estado '`down`' antes.

```
prc (Int, rem) : Proc --> prc (Int, down) .
```

Um processo acessará a região crítica quando o semáforo for zero.

```
    Int' == 0
 -- -----------------------------------------------------
 prc (Int, down) : Proc -{sem = Int', sem' = Int', -}->
 prc (Int, down) .

    Int' > 0, Int'' := Int' - 1
 -- -------------------------------------------------------
 prc (Int, down) : Proc -{sem = Int', sem' = Int'', -}->
 prc (Int, crit) .
```

Ao sair da região crítica, o processo passa pelo estado '`up`', que incrementa em um o valor do semáforo.

```
 prc (Int, crit) : Proc --> prc (Int, up) .

    Int'' := Int' + 1
 -- -----------------------------------------------------
 prc (Int, up) : Proc -{sem = Int', sem' = Int'', -}->
 prc (Int, rem) .
```

Uma busca por uma condição de corrida não é bem sucedida.

```
search : <(prc(0,rem)prc(1,rem))::: 'Soup,{sem = 1}> =>*
         <(prc(0,crit)prc(1,crit))::: 'Soup,R:Record > .

No solution.
```

Vamos usar o verificador de modelos para confirmar este resultado. Começamos criando uma operação auxiliar '`create-conf(i)`' que cria uma configuração com `i` processos. A proposição '`race-condition`' é satisfeita sempre quando mais de um processo estiver em sua região crítica.

```
rewrites: 817190 in 7638ms cpu (7638ms real)
          (106978 rewrites/second)
reduce in CHECK :
  modelCheck(create-conf(10), [] ~ race-condition)
```

```
result Bool :
  true
```

É interessante observar que, já que o sistema não tem justiça, existe uma possibilidade de um processo *nunca* entrar na região crítica. Vamos adicionar uma outra proposição '`in-crit(i)`' que é satisfeita quando um processo `i` estiver em seu estado '`crit`'. A seguinte verificação falha com um contra-exemplo onde o processo '`1`' está "preso" no estado '`down`'.

```
reduce in CHECK :
  modelCheck(create-conf(3), <> in-crit(1))
result ModelCheckResult :
  counterexample

{prc (1, rem) prc (2, rem) prc (3, rem)}
{prc (1, down) prc (2, rem) prc (3, rem)}
{prc (1, down) prc (2, down) prc (3, rem)}
{prc (1, down) prc (2, crit) prc (3, rem)}
{prc (1, down) prc (2, up)  prc (3, rem)}
{prc (1, down) prc (2, up)  prc (3, down)}
{prc (1, down) prc (2, rem) prc (3, down)}
{prc (1, down) prc (2, down) prc (3, down)},
{prc (1, down) prc (2, crit) prc (3, down)}
```

E.2 Filósofos glutões

Esta seção completa a seção 6.3.2.2 com o seguinte material adicional: o restante das regras para a especificação dos Filósofos Glutões; uma variante onde os filósofos comem apenas uma vez; uma outra variante com um escalonamento justo; e uma versão incorreta que é analisada e verificado o *deadlock* com as ferramentas formais de Maude.

E.2.1 Restante das regras

Esta seção mostra o restante das regras da seção 6.3.2.2, ou seja, as regras para os estados '`stest-left`', '`sleave-try`', '`scrit`' e '`srem`'.

A regra para ‘`stest-left`’ é similar à para ‘`stest-right`’. A diferença é que, quando o garfo à esquerda é adquirido, o processo move para ‘`sleave-sty`’.

```
odd (Int),
Pids := lookup (((Int + 1) rem n), Queue),
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids)
         else Pids fi,
Queue' := ((((Int + 1) rem n)) |-> Pids') / Queue,
                                      first (Pids') == Int
-- ------------------------------------------------------
prc (Int, stest-left) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, sleave-try) .
```

Uma vez no estado ‘`sleave-sty`’, um processo entra em sua região crítica.

```
                    odd (Int)
-- ---------------------------------------------
prc (Int, sleave-try) : Proc --> prc (Int, scrit) .
```

Após acessar a sua região crítica, um processo move para o estado ‘`sexit`’.

```
                     odd (Int)
-- ---------------------------------------------
prc (Int, scrit) : Proc -{-}-> prc (Int, sexit) .

   odd (Int)
-- ---------------------------------------------
prc (Int, sexit) : Proc --> prc (Int, sreset-right) .
```

Para colocar o garfo à direita à mesa, ele deve remover-se da fila naquele garfo.

```
odd (Int), Pids := lookup (Int, Queue),
Pids' := remove (Int, Pids),
Queue' := (Int |-> Pids') / Queue
-- --------------------------------
```

```
prc (Int, sreset-right) : Proc
   -{q = Queue, q' = Queue', -}->
              prc (Int, sreset-left) .
```

O mesmo é feito para o garfo à esquerda.

```
odd (Int), Pids := lookup (((Int + 1) rem n), Queue),
Pids' := remove (Int, Pids),
Queue' := ((((Int + 1) rem n)) |-> Pids') / Queue
-- ------------------------------------------------------
prc (Int, sreset-left) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, sleave-exit) .
```

Após ambos os garfos terem sido liberados, o processo vai para o estado '`srem`' (pensando) e volta a ficar com fome.

```
                    odd (Int)
-- ---------------------------------------------
prc (Int, sleave-exit) : Proc --> prc (Int, srem) .
```

```
                   odd (Int)
-- ----------------------------------------
prc (Int, srem) : Proc --> prc (Int, stry) .
```

E.2.2 Filósofos glutões, especificação com término

Esta seção apresenta uma variante da especificação em que cada filósofo come apenas uma vez, imprimindo seu pid quando fizer isto. Esta especificação é inspirada pela presente em [21].

A especificação é similar à exibida na seções 6.3.2.2 e E.2.1 com algumas modificações. A primeira é a adição de um componente apenas-escrita '`Int*`', indexado por '`out'`', para modelar a impressão dos pids.

```
Label = {out' : Int*, q : Queue, q' : Queue, ...} .
```

Vamos mudar a regra para o estado '`scrit`', de forma a fazer o processo emitir o seu pid.

```
                      odd (Int)
 -- ------------------------------------------------------
 prc (Int, scrit) : Proc -{out' = Int, -}-> prc (Int, sexit) .
```

A seguinte regra deve ser removida para um filósofo parar após comer.

```
              odd (Int)
 -- ---------------------------------------
 prc (Int, srem) : Proc --> prc (Int, stry) .
```

Esta pequena modificação no algoritmo abre espaço para algumas verificações interessantes. Buscando por todos os estados finais, chegamos a todos os estados em que o componente '`out`' contém todos os pids de todos os processos na configuração.

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {..., out' = 0,1,2,3}>

Solution 2
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {...,out' = 0,1,3,2}>

Solution 3
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {...,out' = 0,3,1,2}>
...
```

A ordem em que cada filósofo come não é determinística, e então haverá um estado final para cada ordem possível de '`out`'.

Seguindo o exemplo em [21], vamos verificar esta especificação usando uma proposição '`check(i)`' que é satisfeita quando o componente '`out`' contém todos os números menores que `i`.

```
rewrites: 505248 in 2450ms cpu (2440ms real)
```

```
          (206223 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> check (n - 1))
result Bool :
  true
```

Neste caso, como um processo eventualmente pára, todos os processos eventualmente comem. O exemplo abaixo mostra o caso do processo '0'.

```
rewrites: 176389 in 1750ms cpu (1750ms real)
          (100793 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state (0,scrit))
result Bool :
  true
```

E.2.3 Escalonamento justo

É interessante ver o que acontece quando se adiciona um escalonamento justo à especificação, de acordo com a discussão na seção 6.3.1.2. Iremos fazer estas modificações na especificação onde os filósofos param após comer (E.2.2), mas elas são aplicáveis à outra também. Além das regras de escalonamento a única modificação é a adição desta regra.

```
                  odd (Int)
 -- ---------------------------------------
 prc (Int, srem) : Proc --> prc (Int, srem) .
```

Esta regra é necessária para que um processo passe a vez após comer (pois estará parado).

O resultado interessante com esta modificaçaão é que a capacidade de verificação é bem maior. Podemos, por exemplo, verificar uma configuração com 200 filósofos, buscando por um *deadlock*. Note que não há mais estado final agora, já que, após terminarem de comer, os filósofos ficam "passando a vez" indefinidamente.

```
rewrites: 86864 in 10442ms cpu (10501ms real)
          (8318 rewrites/second)
```

```
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

A verificação com a proposição '`check(i)`' também é bem sucedida.

```
rewrites: 1661019 in 42601ms cpu (43419ms real)
          (38989 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> check(n - 1))
result Bool :
  true
```

E.2.4 Uma especificação incorreta

Esta seção mostra como um *deadlock* é detectado usando uma especificação incorreta.

Podemos tornar qualquer uma das especificações exibidas incorreta removendo o subconjunto de regras que se aplicam a processos pares ou ímpares e também o predicado '`odd(i)`' (ou '`even(i)`') das condições das regras.

Vamos começar com a especificação em que cada filósofo fica com fome novamente após pensar. Uma busca por um estado final com uma configuração de quatro filósofos encontra o estado problemático: todos os filósofos estão "presos" segurando seus garfos à esquerda.

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <-
   < prc (0,stest-left) prc (1,stest-left)
     prc (2,stest-left) prc (3,stest-left))
     { q = (0 |->[0] +++ 1 |->[1] +++
            2 |->[2] +++ 3 |->[3]) } >

No more solutions.
```

Com a especificação que termina, a busca encontra, além de todos os estados em que os filósofos comem, o estado problemático ('`Solution 1`').

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <-
  < prc (0, stest-left) prc (1, stest-left)
    prc (2, stest-left) prc (3, stest-left)),
    { fair = 0,out' = (),
      q = (0 |-> [0] +++ 1 |-> [1] +++
           2 |-> [2] +++ 3 |-> [3]) }>

Solution 2
C:Conf <-
  < prc (0, srem) prc (1, srem)
    prc (2, srem) prc (3, srem),
    { fair = 0, out' = 0,1,2,3,
      q =(0 |-> [] +++ 1 |-> [] +++
          2 |-> [] +++ 3 |-> []) }>

Solution 3
C:Conf <-
  < prc (0, srem) prc (1, srem)
    prc (2, srem) prc (3, srem),
    { fair = 0, out' = 0,1,3,2,
      q =(0 |-> [] +++ 1 |-> [] +++
          2 |-> [] +++ 3 |-> []) }>
...
```

A especificação com escalonamento também está sujeita ao *deadlock*.

```
rewrites: 671 in 20ms cpu (20ms real) (33550 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
```

```
C:Conf <-
   < prc (0,stest-left) prc (1,stest-left)
     prc (2,stest-left) prc (3,stest-left)),
     { fair = 0, out' = (),
       q =(0 |->[0]+++ 1 |->[1]+++ 2 |->[2]+++ 3 |->[3]) }>
```

E.3 Algoritmo da Padaria de Lamport

Esta seção descreve uma especificação do algoritmo da Padaria (*Bakery Algorithm*), descrito em [51]. O objetivo é fazer a verificação de um algoritmo com estados infinitos através de uma abstração [65, 66]. Intuitivamente, o algoritmo simula uma padaria onde os consumidores esperam por sua vez, escolhendo tíquetes ao entrarem e são servidos na ordem de seus números escolhidos.

Vamos começar com a descrição formal definindo dois componentes escrita-leitura: '`ch`' modela se um processo está escolhendo seu número ou não; '`nm`' contém o número escolhido. Ambos os componentes são um mapa de inteiros (os pids) para inteiros.

```
IntM = (Int, Int) Map .
Label = {ch : IntM, ch' : IntM,
         nm : IntM, nm' : IntM, ...} .
```

Os processos listados abaixo serão explicados ao longo das regras.

```
St .
St ::= choosing (Int, Int)
     | waiting (Int)
     | rem
     | crit
     | try
     | exit .
```

Quando um processo quer entrar em sua região crítica (entrar na padaria), ele avisa os outros mudando a sua entrada no mapa '`ch`' para '`1`'. O processo então escolhe um número que é maior que os números escolhidos pelos outros processos. Isto é feito no estado '`choosing(i,m)`', em que `i` contém o número dos processos que faltam para checar

e m o maior número encontrado até o momento. Vamos assumir que a constante 'n' será instanciada por uma equação com o número do processos em execução.

```
            IntM' := (Int |-> 1) / IntM
-- ------------------------------------------------
prc (Int, try) : Proc -{ch = IntM, ch' = IntM', -}->
                prc (Int, choosing (n - 1, -1)) .
(Int1 >= 0), (Int1 =/= Int), (Int2' := lookup (Int1, IntM)),

Int3 := if Int2' > Int2 then Int2' else Int2 fi
-- ------------------------------------------------------
prc (Int, choosing (Int1, Int2)) : Proc
                      -{nm = IntM, nm' = IntM, -}->
                      prc (Int, choosing (Int1 - 1, Int3)) .
```

Durante a escolha do número, um processo deve ignorar o seu próprio número.

```
prc (Int, choosing (Int, Int2)) : Proc -->
                prc (Int, choosing (Int - 1, Int2)) .
```

Quando $i = -1$, o maior número encontrado é m. O processo então escolhe $m + 1$ como seu próprio número e vai para a fase seguinte do algoritmo.

```
Int'' := (Int' + 1), IntM'1 := (Int |-> 0) / IntM1,
IntM'2 := (Int |-> Int'') / IntM2
-- ------------------------------------------------
prc (Int, choosing (-1, Int')) : Proc
                -{ch = IntM1, ch' = IntM'1,
                  nm = IntM2, nm' = IntM'2, -}->
                            prc (Int, waiting (0)) .
```

Nesta fase, um processo verifica se chegou a sua vez, passando pelo estado '`waiting(i)`', onde $0 \leq i \leq n-1$. Espera até que o seu número seja o menor de todos os números para acessar sua região crítica. Ele evita comparar com qualquer processo que esteja em fase de escolha.

Já que existe uma possibilidade de que vários processos comecem o processo de escolha ao mesmo tempo, pode acontecer destes processos escolherem o mesmo número. Para lidar com isto, a comparação para encontrar o menor número é feita lexicograficamente usando $(\mathtt{i}, \mathtt{p})$ onde $\mathtt{i}$ é o número do processo e $\mathtt{p}$ seu pid.

```
prc (Int, waiting (Int)) : Proc -->
    prc (Int, waiting ((Int + 1) rem n)) .

  Int' =/= Int,
  (Int1' := lookup (Int', IntM1)),
  (Int2' := lookup (Int', IntM2)),
  (Int2  := lookup (Int, IntM2)),
  St := if Int1' == 0 and
           (Int2' == 0 or
            ((Int2 < Int2') or
             (Int2 == Int2' and Int < Int')))
        then crit else waiting ((Int' + 1) rem n)
        fi
-- ------------------------------------------
prc (Int, waiting (Int')) : Proc
                -{ch = IntM1, ch' = IntM1,
                   nm = IntM2, nm' = IntM2, -}->
                                 prc (Int, St) .
```

Ao sair de sua região crítica, um processo, muda seu número escolhido para zero e move para o estado '`rem`' e volta a tentar acessar novamente a região crítica.

```
                IntM2' := (Int |-> 0) / IntM2
-- ------------------------------------------------------
prc (Int, crit) : Proc -{nm = IntM2, nm' = IntM2', -}->
                                              prc (Int, rem) .

prc (Int, rem) : Proc --> prc (Int, try) .
```

Infelizmente, este algoritmo não estabelece um limite no número escolhido. Além disto, uma solução aparentemente trivial de usar inteiros *modulo* algum valor $\mathtt{b}$ grande também falha.

Podemos verificar que não existe limite superior no número escolhido usando uma busca que, para dois processos, mostra que um deles facilmente escolhe o número dez (ou qualquer outro natural). O problema acontece quando um processo escolhe um número enquanto o outro está na sua região crítica. Um processo apenas zera o seu número escolhido *após* sair da região crítica. O problema acontece da seguinte maneira: o processo '0', com um número escolhido 2 está em sua região crítica; processo '1' escolhe 3 como seu número; quando este processo entra na sua região crítica, processo '0' escolhe um outro número, que será 4, e assim por diante.

```
search [1] in BAKERY : initial-conf =>*
    < S:Soup,{PR:PreRecord,n = (0 |-> 10 +++ 1 |-> I:Int)} > .

...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 2 +++ 1 |-> 1)} >
< (prc(0, waiting(1)) prc(1, rem)),
  {n = (0 |-> 2 +++ 1 |-> 0)} >
...
< (prc(0, crit) prc(1, choosing(-1, 2))),
  {n = (0 |-> 2 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 4 +++ 1 |-> 3)} >
...
< (prc(0, crit) prc(1, choosing(-1, 4))),
  {n = (0 |-> 4 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 6 +++ 1 |-> 5)} >
...
< (prc(0, crit) prc(1, choosing(-1, 6))),
   {n = (0 |-> 6 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 8 +++ 1 |-> 7)} >
...
```

```
< (prc(0, crit) prc(1, choosing(-1, 8))),
  {n = (0 |-> 8 +++ 1 |-> 0)} >
...
< (prc(0, choosing(1, -1)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, choosing(0, 9)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, choosing(-1, 9)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, waiting(0)) prc(1, waiting(0))),
  {n = (0 |-> 10 +++ 1 |-> 9)} >
```

Modificando o algoritmo de forma que o número escolhido seja *modulo*, digamos, 2267, o problema também ocorre.

```
search [1] in BAKERY : initial-conf =>*
    < (prc(0, crit) prc(1, crit)) ::: 'Soup,{PR:PreRecord} > .

Solution 1 (state 183534)
states: 183535  rewrites: 5607219 in 34110ms cpu
    (34130ms real) (164386 rewrites/second)
PR:PreRecord --> ch = (0 |-> 0 +++ 1 |-> 0),
                      n = (0 |-> 2266 +++ 1 |-> 0)

...
< (prc(0, try) prc(1, choosing(-1, 2264))) ,
  {n = (0 |-> 0 +++ 1 |-> 0)} >
< (prc(0, choosing(1, -1)) prc(1, choosing(-1, 2264))),
  {n = (0 |-> 0 +++ 1 |-> 0)} >
< (prc(0, choosing(1, -1)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, choosing(0, 2265)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, choosing(-1, 2265)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, waiting(0)) prc(1, waiting(0))),
```

```
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
< (prc(0, waiting(1)) prc(1, waiting(0))),
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
< (prc(0, waiting(1)) prc(1, rem)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, rem)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, try)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(1, -1))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(0, -1))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(-1, 2266))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, waiting(0))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, crit)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
```

Para que este algoritmo seja passível de verificação, devemos criar uma abstração que captura a essência do algoritmo, mas não contém o número infinito de estados do original. A solução segue as idéias em [65] no qual uma abstração para dois processos é definida e uma prova da sua corretude é dada.

A chave para se encontrar a abstração correta neste caso é perceber que o valor absoluto não é importante, mas o seu valor relativo em relação aos demais.

Começamos com as duas equações abaixo: um processo muda o seu número para zero após sair da região crítica; então o número escolhido pelo outro processo não precisa crescer indefinidamente: o número um é suficiente.

```
ceq (< S:Soup, { n = (0 |-> 0 +++ 1 |-> I), PR } >)
  = < S:Soup, { n = (0 |-> 0 +++ 1 |-> 1), PR } >
if I > 1 .
```

```
ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> 0), PR } >)
  = < S:Soup, { n = (0 |-> 1 +++ 1 |-> 0), PR } >
if I > 1 .
```

Em seguida, as seguintes equações controlam os números escolhidos pelos processos ao mesmo tempo em que mantêm seus valores relativos.

```
ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> I'), PR } >)
  = < S:Soup, { n = (0 |-> 2 +++ 1 |-> 1), PR } >
if (I' < I) /\ not (I' == 1 and I == 2) .

ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> I'), PR } >)
  = (< S:Soup, { n = (0 |-> 1 +++ 1 |-> 1), PR } >)
if not (I' < I) /\ not (I' == 1 and I == 1) .
```

Com estas abstrações, podemos tentar buscar por uma condição de corrida.

```
rewrites: 4195 in 61ms cpu (61ms real) (67671 rewrites/second)
search in CHECK :
   initial-conf =>* <(prc(0,crit)prc(1,crit))::: 'Soup,
   {PR:PreRecord}> .

No solution.
```

Além disto, ambos os processos eventualmente atingem sua região crítica, de acordo com as duas buscas abaixo:

```
rewrites: 3463 in 36ms cpu (36ms real) (93609 rewrites/second)
search in CHECK :
   initial-conf =>* <(prc(0,crit)prc(1,St:St))::: 'Soup,
   {PR:PreRecord}> .

Solution 1
PR:PreRecord <- ch =(0 |-> 0 +++ 1 |-> 0),
                n =(0 |-> 1 +++ 1 |-> 1);
```

```
St:St <- waiting(0)
rewrites: 3376 in 26ms cpu (26ms real) (125055 rewrites/second)

search in CHECK :
   initial-conf =>* <(prc(1,crit)prc(0,St:St))::: 'Soup,
   {PR:PreRecord}> .

Solution 1
PR:PreRecord <- ch =(0 |-> 0 +++ 1 |-> 0),
                n =(0 |-> 2 +++ 1 |-> 1);
St:St <- waiting(0)
Bye.
```

E.4 Eleição de líder num anel assíncrono

Esta seção especifica um algoritmo para eleição de líder num anel assíncrono e unidirecional. É usado como exemplo de uma especificação que usa o modelo de troca de mensagens e faz uso de alguns exemplos mais complexos de verificação de modelos. A idéia intuitiva é eleger como líder o processo que tenha o maior pid de todos os processos no anel. Cada processo encaminha o seu próprio pid para o seu vizinho; um processo, ao receber um pid que é maior do que o seu, o encaminha para seu vizinho novamente. Quando um processo receber o seu próprio pid, ele sabe que é o líder.

Vamos iniciar a descrição formal do algoritmo com a definição do formato da mensagem. Ela contém como primeiro argumento um pid e como segundo argumento o destino da mensagem. Não há necessidade de saber a origem da mensagem, já que estamos lidando com uma topologia conhecida.

```
Msg ::= m Int to Int .
```

A rede em anel é modelada fazendo com que cada processo saiba o pid de seu vizinho. Apenas um vizinho é conhecido, já que a comunicação é unidirecional.

```
Proc ::= prc (Int, Int', St) .
```

Os estados são:

```
St ::= start
     | waiting
     | leader .
```

No início, cada processo envia seu pid para seu vizinho.

```
prc (Int, Int', start) : Soup -->
               prc (Int, Int', waiting) (m Int to Int') .
```

Quando um processo recebe uma mensagem, ele compara o pid $\mathtt{i}'$ recebido com o seu, $\mathtt{i}$. Se $\mathtt{i}' > \mathtt{i}$, ele encaminha a mensagem para o vizinho.

```
   Int'' > Int
-- ---------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
            prc (Int, Int', waiting) (m Int'' to Int') .
```

Caso contrário ($\mathtt{i}' < \mathtt{i}$), a mensagem é "consumida."

```
   Int'' < Int
-- ---------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
            prc (Int, Int', waiting) .
```

Quando $\mathtt{i}' = \mathtt{i}$, o processo sabe que é o líder.

```
   Int'' == Int
-- ---------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
                                prc (Int, Int', leader) .
```

Para verificar a corretude da especificação, vamos fazer algumas verificações de modelos com uma configuração de quatro processos. Começamos criando uma operação '`leaders(S)`' que computa o número de líderes na sopa `S`.

```
op leaders : Soup -> Int .

eq leaders (S S') = leaders (S) + leaders (S') .
eq leaders (prc (I, I', leader)) = 1 .
eq leaders (prc (I, I', waiting)) = 0 .
eq leaders (prc (I, I', start)) = 0 .
eq leaders (m I to I') = 0 .
```

A proposição '`one-leader`' é satisfeita quando existe exatamente um único líder na sopa, e '`no-leader`' é satisfeita quando não há líderes.

```
op one-leader : -> Prop .
op no-leader : -> Prop .

eq < S ::: 'Soup, R > |= one-leader = (leaders (S) == 1) .
eq < S ::: 'Soup, R > |= no-leader = (leaders (S) == 0) .
```

Podemos verificar a seguinte fórmula: em todas as execuções, sempre existirá um único líder.

```
rewrites: 7339322 in 56444ms cpu (56447ms real)
          (130027 rewrites/second)
reduce in CHECK :
  modelCheck(init, <> [] one-leader)
result Bool :
  true
```

Não existe execução em que nenhum líder é eleito.

```
rewrites: 7204804 in 56551ms cpu (57278ms real)
          (127402 rewrites/second)
reduce in CHECK :
  modelCheck(init, ~ [] no-leader)
result Bool :
  true
```

Em todas as execuções, não existem líderes até que um seja eleito.

```
rewrites: 7340679 in 57867ms cpu (58915ms real)
          (126853 rewrites/second)
reduce in CHECK :
  modelCheck(init, [] (no-leader U one-leader))
result Bool :
  true
```

APÊNDICE F – Lógica combinatória

O objetivo deste capítulo é demonstrar que, dependendo das características da especificação, o interpretador Maude pode atingir velocidades da ordem de 10^6 reescritas/segundo. Como um exemplo, vamos especificar um interpretador simples de lógica combinatória. A literatura sobre o assunto é vasta, e optamos por seguir [67].

Começamos pelos combinadores primitivos S and K.

```
fmod CL is
 sort Exp .

 op S : -> Exp [ctor] .
 op K : -> Exp [ctor] .

 op __ : Exp Exp -> Exp [gather(E e)] .

 vars x y z : Exp .

 eq K x y = x .
 eq S x y z = x z (y z) .
endfm
```

O módulo 'CL-EXT' estende este conjunto básico com os combinadores de Curry, B, C e I, definido em termos dos combinadores S e K de Shönfinkel.

```
fmod CL-EXT is including CL .
 op B : -> Exp .  eq B = ((S(K S))K) .
 op C : -> Exp .  eq C = ((S(K((S S)(K K))))((S(K K))S)) .
 op I : -> Exp .  eq I = ((S K)K) .
endfm
```

Este conjunto combinado é usado para definir os inteiros positivos, conforme a seguinte progressão mostra:

$$\begin{aligned} 1 &\equiv ((\mathsf{SB})(\mathsf{KI})) \\ 2 &\equiv ((\mathsf{SB})((\mathsf{SB})(\mathsf{KI}))) \\ 3 &\equiv ((\mathsf{SB})((\mathsf{SB})((\mathsf{SB})(\mathsf{KI})))) \\ &\ldots \end{aligned}$$

O módulo '`CL-NATURALS`' implementa esta idéia com o operador '`$(n)`' que converte o natural `n` para a sua expressão equivalente. Vamos também definir algumas operações adicionais que implementam adição, multiplicação, e exponenciação: respectivamente, '`pl`', '`ti`' e '`ex`'.

```
fmod CL-NATURALS is including CL-EXT .
 op $ : Nat -> Exp .

 var n : Nat .

 eq $(s(n)) = (S B) $(n) .
 eq $(0) = (K I) .

 op pl : -> Exp .
 op ti : -> Exp .
 op ex : -> Exp .

 eq pl = ((C I) (S B)) .
 eq ti = ((B((C C)(K I)))((C B) pl)) .
 eq ex = (C((B(C((C C)((S B)(K I))))) ti)) .
endfm
```

Com '`CL-NATURALS`' é possível calcular, por exemplo, $1 + 1$.

```
Maude> red pl $(1) $(1) .
reduce in CL-NATURALS : pl $(1) $(1) .
rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
result Exp: S (S (K S) K) (S (S (K S) K) (K (S K K)))
```

Para converter esta seqüência de volta em numerais usamos o fato de que os números, quando visto como combinadores, têm uma característica interessante: se ε_1 e ε_2 são expressões, e $\hat{\mathfrak{n}}$ é a expressão de combinadores equivalente ao número $\mathfrak{n}$, então, $\hat{1}\varepsilon_1\varepsilon_2 = \varepsilon_1\varepsilon_2, \hat{2}\varepsilon_1\varepsilon_2 = \varepsilon_1\varepsilon_1\varepsilon_2, \hat{3}\varepsilon_1\varepsilon_2 = \varepsilon_1\varepsilon_1\varepsilon_1\varepsilon_2, \ldots$ O módulo '`NATURALS-CL`' implementa esta idéia.

```
fmod NATURALS-CL is including CL-NATURALS .
 vars x y : Exp .

 ops eqv eqv-aux : Exp -> Nat .

 ops i j : -> Exp .

 eq eqv (x) = eqv-aux (x i j) .
 eq eqv-aux (x y) = eqv-aux(x) + eqv-aux(y) .
 eq eqv-aux (i) = 1 .
 eq eqv-aux (j) = 0 .
endfm
```

Finalmente, a seguinte redução atinge a velocidade de um milhão de reescritas por segundo. Isto é certamente causado pelo grande número de padrões repetidos de S e K no operador de justaposição.

```
Maude> reduce in NATURALS-CL : eqv(ti $(500) $(500)) .
reduce in NATURALS-CL : eqv(ti $(500) $(500)) .
rewrites: 2506069 in 2178ms cpu (2184ms real)
   (1150275 rewrites/second)
result NzNat: 250000
```

Referências

[1] PLOTKIN, G. D. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, v. 60–61, p. 17–139, 2004. Special issue on SOS.

[2] REPPY, J. CML: A higher-order concurrent language. In: SIGPLAN. *Programming Language Design and Implementation*. [S.l.]: ACM, 1991. p. 293–259.

[3] MILNER, R. et al. *The definition of Standard ML (Revised)*. [S.l.]: MIT Press, 1997.

[4] MILNER, R. *A Calculus of Communicating Systems*. [S.l.]: Springer-Verlag, 1980. (Lecture Notes in Computer Science, v. 92).

[5] MILNER, R. *Communicating and Mobile Systems: the π-Calculus*. [S.l.]: Cambridge University Press, 1999.

[6] HENNESSY, M. *A Semantics of Programming Languages: An Elementary Introduction Using Operational Semantics*. [S.l.]: John Wiley and Sons, 1990. Currently out of print; available from `http://www.cogs.susx.ac.uk/users/matthewh/semnotes.ps.gz`.

[7] NIELSON, H. R.; NIELSON, F. *Semantics with Applications: A Formal Introduction*. Chichester, England: John Wiley & Sons, 1992. 240 p. (Wiley Professional Computing). ISBN 0-471-92-80-8.

[8] REYNOLDS, J. C. *Theories of Programming Languages*. Cambridge, England: Cambridge University Press, 1998.

[9] PIERCE, B. C. *Types and Programming Languages*. [S.l.]: MIT Press, 2002.

[10] MOSSES, P. D. Fundamental Concepts and Formal Semantics of Programming Languages—an introductory course. Lecture notes, available at `http://www.daimi.au.dk/jwig-cnn/dSem/`. 2004.

[11] MOSSES, P. D. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, v. 60–61, p. 195–228, 2004. Special issue on SOS.

[12] MESEGUER, J. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, Elsevier, v. 96, n. 1, p. 73–155, April 1992.

[13] MESEGUER, J. Membership algebra as a semantic framework for equational specification. In: PARISI-PRESICCE, F. (Ed.). *WADT'97*. [S.l.]: Springer, 1998. v. 1376, p. 18–61.

[14] DENKER, G.; MESEGUER, J.; TALCOTT, C. Protocol specification and analysis in Maude. In: *Heintze, N. and Wing, J., editors, Proc. of Workshop on Formal Methods and Security Protocols*. [S.l.: s.n.], 1998. Indianápolis, Indiana.

[15] BRAGA, C. et al. Maude Action Tool: Using reflection to map action semantics to rewriting logic. In: RUS, T. (Ed.). *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*. [S.l.]: Springer-Verlag, 2000. (Lecture Notes in Computer Science, v. 1816), p. 407–421.

[16] DENKER, G.; MESEGUER, J.; TALCOTT, C. Formal specification and analysis of active networks and communication protocols: The Maude experience. In: *DISCEX 2000, Proc. Darpa Information Survivability Conference and Exposition, Hilton Head, South Carolina*. [S.l.]: IEEE Computer Society Press, 2000. v. 1, p. 251–265.

[17] ÖLVECZKY, P. C. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. Tese (Doutorado) — University of Bergen, Norway, 2000. `http://maude.csl.sri.com/papers`.

[18] BRAGA, C. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, September 2001. `http://www.ic.uff.br/~cbraga`.

[19] STEHR, M.-O.; SRIDHARANARAYANAN, A. Formal specification of sectrace. In: *Workshop on Context Sensitive Systems Assurance (Contessa'03)*. [S.l.: s.n.], 2003.

[20] MESEGUER, J.; BRAGA, C. Modular rewriting semantics of programming languages. In: RATTRAY, C.; MAHARAJ, S.; SHANKLAND, C. (Ed.). *In Algebraic Methodology and Software Technology: proceedings of the 10th International Conference, AMAST 2004*. Stirling, Scotland, UK: Springer, 2004. (LNCS, v. 3116), p. 364–378. ISSN 0302-9743, ISBN 3-540-22381-9.

[21] FARZAN, A. et al. Formal analysis of Java programs in JavaFAN. In: ALUR, R.; PELED, D. A. (Ed.). *CAV*. [S.l.]: Springer, 2004. (Lecture Notes in Computer Science).

[22] RADEMAKER, A.; BRAGA, C.; SZTAJNBERG, A. A rewriting semantics for a software architecture description language. 2005. To appear.

[23] MARTÍ-OLIET, N.; MESEGUER, J. Handbook of philosophical logic. In: _____. Second. [S.l.]: Kluwer Academic Publishers, 2001. v. 61, cap. Rewriting Logic as a Logical and Semantic Framework. `http://maude.cs.uiuc.edu/papers`.

[24] VERDEJO, J. A. *Maude como um marco semântico ejecutable*. Tese (Doutorado) — Universidad Complutense Madrid, 2003.

[25] BRAGA, C.; MESEGUER, J. Modular rewriting semantics in practice. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 117, p. 393–416, 2005.

[26] CLAVEL, M. et al. *Maude Manual (Version 2.1)*. `http://maude.cs.uiuc.edu`, March 2004.

[27] ACETO, L.; FOKKINK, W. J.; VERHOEF, C. *Conservative Extension in Structural Operational Semantics*. Department of Computer Science, Institute of Electronic Systems, Aalborg University, September 1999. 23 pp. Appears in the *Bulletin of the European Association for Theoretical Computer Science*, 70:110–132, 1999.

[28] HARTEL, P. H. LETOS – A lightweight execution tool for operational semantics. *Software—Practice and Experience*, v. 29, n. 15, p. 1379–1416, Sep 1999. Disponível em: <http://www.ecs.soton.ac.uk/ phh/letos.html>.

[29] BOROVANSKÝ, P. et al. Elan from a rewriting logic point of view. *Theoretical Computer Science*, v. 285, p. 155–185, 2002.

[30] FUTATSUGI, K.; DIACONESCU, R. Cafeobj report. *World Scientific, AMAST Series*, 1998.

[31] CLAVEL, M.; MESEGUER, J.; PALOMINO, M. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. In: GADDUCCI, F.; MONTANARI, U. (Ed.). *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*. [S.l.]: Elsevier, 2002. (Electronic Notes in Theoretical Computer Science, v. 71).

[32] BRUNI, R.; MESEGUER, J. Generalized rewrite theories. In: *Thirtieth International Colloquium on Automata, Languages and Programming*. [S.l.]: Springer-Verlag, 2003. (Lecture Notes in Computer Science).

[33] MESEGUER, J. Software specification and verification in rewriting logic. Lectures given at the NATO Advanced Study Institute International Summer School, Marktoberdorf, Germany, 2002. Available from `http://maude.cs.uiuc.edu`. 2003.

[34] MESEGUER, J.; ROŞU, G. Rewriting logic semantics: From language specifications to formal analysis tools. In: BASIN, D.; RUSINOWITCH, M. (Ed.). *Proceedings of the 2nd International Joint Conference on Automated Reasoning, IJCAR'04, (Cork, Ireland)*. [S.l.]: Springer, 2004. (Lecture Notes in Computer Science, v. 3097), p. 183–197.

[35] VIRY, P. Equational rules for rewriting logic. *Theor. Comput. Sci.*, v. 285, n. 2, p. 487–517, 2002.

[36] CLAVEL, M. et al. *Maude: Specification and Programming in Rewriting Logic*. `http://maude.csl.sri.com`, January 1999.

[37] MARTÍ-OLIET, N.; MESEGUER, J. *Rewriting Logic as a Logical and Semantic Framework*. [S.l.], August 1993. To appear in D. Gabbay, editor, *Handbook of Philosophical Logic, Second Edition, Volume 6*, Kluwer Academic Publishers, 2001. `http://maude.csl.sri.com/papers`.

[38] CLAVEL, M. et al. Maude as a formal meta-tool. In: WING, J. M.; WOODCOCK, J.; DAVIES, J. (Ed.). *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20–24, 1999 Proceedings, Volume II*. [S.l.]: Springer-Verlag, 1999. (Lecture Notes in Computer Science, v. 1709), p. 1684–1703.

[39] CLAVEL, M. et al. Maude as a metalanguage. In: KIRCHNER, C.; KIRCHNER, H. (Ed.). *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*. [S.l.]: Elsevier, 1998. (Electronic Notes in Theoretical Computer Science, v. 15), p. 237–250. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[40] GOGUEN, J. et al. An introduction to obj3. In: KAPLAN, S.; JOUANNAUD, J.-P. (Ed.). *Conditional Term Rewriting Systems, 1st International workshop, Orsay, France*. New York, NY: Springer-Verlag, 1987, (Lecture Notes in Computer Science, v. 308). p. 258–263.

[41] GOGUEN, J. A.; MALCOLM, G. More higher order programming in OBJ3. In: GOGUEN, J. A.; MALCOLM, G. (Ed.). *Software Engineering with OBJ: Algebraic Specification in Action*. [S.l.]: Kluwer, Boston, 2000. cap. 9, p. 397–408.

[42] MESEGUER, J.; BRAGA, C. Modular rewriting semantics of programming languages. Manuscript. `http://maude.cs.uiuc.edu/papers`.

[43] PETTERSSON, M. *Compiling Natural Semantics*. [S.l.]: Springer, 1999. (Lecture Notes in Computer Science, v. 1549).

[44] TURNER, D. A. Miranda: A non-strict functional language with polymorphic types. In: JOUANNAUD, J. (Ed.). *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architectures, Nancy, France*. New York, NY: Springer-Verlag, 1985, (Lecture Notes in Computer Science, v. 201). p. 1–16.

[45] JONES, S. P. (Ed.). *Haskell 98 Language and Libraries; The Revised Report*. [S.l.]: Cambridge University Press, 2003.

[46] KAHN, G. Natural semantics. In: *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*. [S.l.]: Springer-Verlag, 1987, (Lecture Notes in Computer Science, v. 247). p. 22–39.

[47] KNUTH, D. E. *Selected Papers on Computer Languages*. Stanford, CA, USA: CSLI Publications, 2002. xvi + 594 p. ISBN 1-57586-381-2 (hardback), 1-57586-382-0 (paperback).

[48] GOSLING, J.; JOY, B.; STEELE, G. L. *The Java Language Specification*. Reading, MA, USA: Addison-Wesley, 1996. xxv + 825 p., 7.38in x 9.19in x 1.25in. (The Java Series). ISBN 0-201-63451-1. Disponível em: <http://www.aw.com/cp/javaseries.html; http://www.aw.com/cseng/titles/0-201-63451-1/>.

[49] MESEGUER, J.; MARTÍ-OLIET, N.; VERDEJO, A. Towards a strategy language for Maude. In: MARTÍ-OLIET, N. (Ed.). *Proceedings of 5th International Workshop on Rewriting Logic and its Applications, WRLA 2004*. [S.l.]: Elsevier, 2005. (Eletronic Notes in Theoretical Computer Science, v. 117), p. 417–441.

[50] GOGUEN, J. A. et al. Initial algebra semantics and continuous algebras. *Journal of the ACM*, v. 24, n. 1, p. 68–95, 1977.

[51] LYNCH, N. *Distributed Algorithms*. [S.l.]: Morgan Kaufmann, 1996.

[52] DOH, K.-G.; MOSSES, P. D. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, v. 47, n. 1, p. 3–36, April 2003.

[53] APPEL, A. W. *Modern Compiler Implementation in Java: Basic Techniques*. Cambridge, UK: Cambridge University Press, 1997. x + 398 p. ISBN 0-521-58387-X (hardback), 0-521-58654-2 (paperback). Disponível em: <http://www.cs.princeton.edu/ appel/modern/java>.

[54] KAMIN, S. N.; REDDY, U. S. Two semantic models of object-oriented languages. In: GUNTER, C. A.; MITCHELL, J. C. (Ed.). *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. [S.l.]: MIT Press, 1994. p. 464–495.

[55] REDDY, U. S. Objects as closures: Abstract semantics of object oriented languages. In: *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*. Snowbird, Utah: [s.n.], 1988. p. 289–297.

[56] THORUP, L.; TOFTE, M. Object-oriented programming and Standard ML. In: REPPY, J. H. (Ed.). *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Orlando, Florida*. [S.l.]: INRIA, 1994. (Rapport de recherche, 2265), p. 41–49.

[57] UNGAR, D.; CHAMBERS, C. Self: The power of simplicity. In: *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*. [S.l.]: ACM, 1987. p. 227–242.

[58] MESEGUER, J. *Rewriting as a Unified Model of Concurrency*. [S.l.], February 1990. Revised June 1990. Appendices on functorial semantics have not been published elsewhere.

[59] BERRY, G.; BOUDOL, G. The chemical abstract machine. In: *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*. New York: ACM Press, 1990. p. 81–94.

[60] HAXTHAUSEN, A. E.; NICKL, F. Pushouts of order-sorted algebraic specifications. *Lecture Notes in Computer Science*, v. 1101, p. 132–147, 1996. ISSN 0302-9743.

[61] ROsU, G. From conditional to unconditional rewriting. In: *Proc. 17th Int. Workshop on Algebraic Development Techniques (WADT 2004)*. [S.l.: s.n.], 2004.

[62] FELLEISEN, M. *The Calculi of λ-ν-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. Tese (Doutorado) — Indiana University, 1987.

[63] CHALUB, F.; BRAGA, C. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, v. 10, n. 7, p. 789–807, jul. 2004. `http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics`.

[64] BAKER-FINCH, C.; KING, D. J.; TRINDER, P. An operational semantics for parallel lazy evaluation. In: *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM Press, 2000. p. 162–173. ISBN 1-58113-202-6.

[65] MESEGUER, J.; PALOMINO, M.; MARTÍ-OLIET, N. Notes on model checking and abstraction in rewriting logic. Http://maude.cs.uiuc.edu/.

[66] MESEGUER, J.; PALOMINO, M.; MARTÍ-OLIET, N. Equational abstractions. In: BAADER, F. (Ed.). *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*. [S.l.]: Springer-Verlag, 2003. (Lecture Notes in Computer Science, v. 2741).

[67] BRAINERD, W. S.; LANDWEBER, L. H. *Theory of Computation*. New York: John Wiley and Sons, 1974.