A Linguagem de Especificação Formal Machĭna 2.0

Roberto da Silva Bigonha
Fabio Tirelo
Vladimir Oliveira Di Iorio
Mariza Andrade da Silva Bigonha
Mário Celso Candian Lobato

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

Belo Horizonte

27 de Outubro de 2006

Conteúdo

Capítulo 1

Máquinas de Estado Abstratas

Este capítulo trata da definição de Máquinas de Estados Abstratas definidas por Yuri Gurevich em [21, 23] com um conceito poderoso e elegante para modelagem matemática de sistemas dinâmicos discretos, e de sua extensão para tratar sistemas multiagentes.

1.1 Introdução

A idéia original era prover semântica operacional para algoritmos elaborando a tese implícita de Turing: "todo algoritmo é simulado por uma máquina de Turing apropriada". Desta maneira, Turing deu semântica operacional para algoritmos. Entretanto, esta semântica é muito inconveniente, pois pode ser necessária uma longa seqüência de passos na máquina de Turing para simular um único passo do algoritmo. Assim, as ASM foram criadas com o objetivo de simular algoritmos de maneira mais próxima e natural [21]. A Tese de ASM Seqüencial diz que todo algoritmo seqüencial, em qualquer nível de abstração, pode ser visto como uma ASM [24].

A metodologia ASM provê recursos expressivos para especificar a semântica operacional de sistemas dinâmicos discretos, em um nível de abstração natural e de uma maneira direta e essencialmente livre de codificação [17]. Com isso, tem-se por objetivo diminuir a distância que há entre modelos formais de computação e métodos práticos de especificação [23]. Neste sentido, definem-se máquinas de estado que simulem passo a passo o algoritmo. A metodologia utiliza conceitos simples e bem conhecidos, o que facilita a leitura e a escrita de especificações de sistemas. Além disso, ela pode ser utilizada inclusive por novatos e por práticos em computação que possuam pouca base matemática.

Na literatura, há vários exemplos de utilização de ASM na especificação formal de sistemas, dentre os quais podem-se citar: arquiteturas de computadores [7, 8, 11], linguagens de programação [12, 13, 14, 25, 38], sistemas distribuídos [2, 3, 9, 26] e de tempo real [19, 27], entre outros [10].

1.2 Álgebra Evolutiva

Em linhas gerais, ASM são máquinas abstratas, em que um estado é formado por funções e relações. Estas funções ou relações possuem nomes e são definidas em um conjunto denominado o superuniverso do estado. O conjunto dos nomes de funções e relações de um estado é o vocabulário do estado. A interpretação de um nome de função ou relação é um mapeamento dos nomes pertencentes ao vocabulário nas respectivas funções ou relações. A mudança de estado da máquina (transição) é dada por uma regra de transição. Uma regra de transição modifica a interpretação de alguns nomes de função do vocabulário do estado. Uma regra de transição de ASM tem a aparência de um programa como de uma linguagem imperativa comum. A diferença principal é a ausência de iteração, pois este conceito está implícito na execução da máquina. Com efeito, a execução da máquina consiste em executar a sua regra de transição repetidas vezes, modificando a cada vez o estado atual. Desta forma, a execução é formada por uma seqüência de estados de mesmo vocabulário e compostos por diferentes funções e relações. Diz-se que a interpretação dos nomes pertencentes ao vocabulário é modificada de estado para estado durante a execução.

Mais precisamente, um *vocabulário* Υ (pronuncia-se como "Y") é um conjunto de nomes de funções e relações, cada nome com uma aridade fixa associada. Por exemplo, fixando-se a aridade da função de nome i em 0 e da função de nome f em 1, tem-se que o conjunto $\{i, f\}$ é um vocabulário. Os nomes de relações de zero argumento *true*, *false*, o nome de função de zero argumento *undef*, os operadores booleanas usuais e o sinal de igualdade estão presentes em todo vocabulário.

Um *estado* S de vocabulário Υ é um conjunto X, denominado o *superuniverso* de S, junto com as interpretações, em X, dos nomes de funções e relações pertencentes a Υ. Se f é um nome de função de aridade r, então f é interpretado como uma função $\boldsymbol{f} : X^r \rightarrow X$. Se f é um nome de relação de aridade r, então f é interpretado como uma função $\boldsymbol{f} : X^r \rightarrow \{false, true\}$. Se U é um nome de relação pertencente a Υ, então o conjunto $U = \{\bar{x} : U(\bar{x}) = true\}$ é um *universo* contido em X. Desta maneira, diz-se que $U(\bar{x}) = true$ e $\bar{x} \in U$ são proposições equivalentes.

Um novo estado é criado a partir do estado atual, por uma regra de transição, por meio da mudança da interpretação de cada nome de função. As regras mais simples (regras básicas) são *atualização*, *bloco* e *condicional*.

Uma regra de atualização tem a forma $R \equiv f(\bar{x}) := y$, onde o comprimento de $\bar{x}$ é igual à aridade de f. Esta regra cria, a partir de um estado S, um novo estado S', tal que a interpretação do nome de função f é uma função $\boldsymbol{f} : X^r \rightarrow X$, que, no ponto $\bar{\boldsymbol{x}}$ (a avaliação da tupla $\bar{x}$), o seu valor é $\boldsymbol{y}$ (a avaliação de y). Por exemplo, a regra $f(1) := 2$ determina um novo estado no qual o valor da função $\boldsymbol{f}$, no ponto 1, é 2.

Uma regra condicional da forma $R \equiv$ **if** g **then** R_1 **else** R_2 **end** tem a seguinte semântica: se a expressão g avaliar em *verdadeiro*, então o estado resultante é o resultado da regra R_1; caso contrário, o estado resultante é o resultado da regra R_2.

Estado	0	1	2	3	4	$\cdots$	n
S_0	1	*undef*	*undef*	*undef*	*undef*	$\cdots$	*undef*
S_1	1	1	*undef*	*undef*	*undef*	$\cdots$	*undef*
S_2	1	1	2	*undef*	*undef*	$\cdots$	*undef*
S_3	1	1	2	6	*undef*	$\cdots$	*undef*
S_4	1	1	2	6	24	$\cdots$	*undef*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
S_n	1	1	2	6	24	$\cdots$	$n!$

Figura 1.1: Interpretação do nome de função *fat* nos estados que formam a execução da ASM do Fatorial.

Uma regra bloco da forma $R \equiv R_1, \ldots, R_n$ tem a seguinte semântica: o estado formado por R é o resultado da execução de todas as regras R_i em paralelo. Por exemplo, a execução da regra bloco

$$f(1) := 2, f(2) := 4$$

produz um novo estado, no qual o valor da função $\boldsymbol{f}$, no ponto 1, é 2 e, no ponto 2, é 4.

Uma especificação ASM contém a definição de um estado inicial, S_0 e uma regra, R, que define as mudanças de estado. A execução de uma especificação é uma seqüência de estados $\langle S_n : n \geq 0 \rangle$, onde um estado S_i é obtido executando a regra R em S_{i-1}.

Para exemplificar estes conceitos, apresenta-se uma especificação ASM para a função fatorial.

Exemplo 1.2.1 (Função Fatorial) Suponha que o estado inicial S_0 seja dado pelo vocabulário $\Upsilon = \{f, i\}$, onde i é um nome de função de zero argumento, interpretado como 1, e f é um nome de função unária, interpretado como a função $\boldsymbol{f} = \lambda x.x = 0 \rightarrow 1, \mathit{undef}$[1].

A regra da especificação é:

$$f(i) := i \times f(i-1), i := i + 1$$

Ao executar a regra no estado S_0, obtém-se um novo estado S_1, no qual i é interpretado como 2 e f é interpretado como a função

$$\boldsymbol{f} = \lambda x.x = 0 \rightarrow 1, x = 1 \rightarrow 1, \mathit{undef}$$

Como pode-se ver na Figura 1.1, ao executar a regra no estado S_1, obteremos um novo estado S_2, no qual i é interpretado como 2 e a interpretação de f, no ponto 2, passa a ser igual a 2. Da mesma forma, obtemos os estados $S_3, S_4, \ldots$ Intuitivamente, podemos perceber que, no estado S_k, $k \geq i$, a interpretação de f, $\boldsymbol{f}$, é uma função que, aplicada a $\boldsymbol{i}$, retorna $\boldsymbol{i}!$. □

[1] Se $f = \lambda x.E$, então f é uma função dada por $f(x) = E$. Mais detalhes sobre a notação lambda pode ser encontrada em [32]. A notação $a \rightarrow b, c$, utilizada em [20], é equivalente a **if** a **then** b **else** c.

Um aspecto importante que deve ser modelado na especificação de um sistema é que, em geral, sistemas são afetados pelo ambiente. O modelo ASM supõe que o ambiente se manifeste por meio de funções $e_1, \ldots, e_k$, denominadas *funções externas*. Um exemplo típico de função externa é uma entrada fornecida pelo usuário. Pode-se pensar em funções externas como *oráculos*, tais que, a especificação fornece argumentos e o oráculo fornece o resultado [23].

Além das regras básicas – atualização, bloco e condicional – há também as regras que utilizam variáveis[2]. Em ASM, variáveis são utilizadas para modelar paralelismo, não-determinismo e a "criação"de novos elementos. As regras que utilizam variáveis são as regras *import*, *choose* e *forall*.

Regra Import

A regra *import* tem a forma

$$\textbf{import } v \textbf{ do } R_0 \textbf{ end},$$

onde v é uma variável e R_0 é uma regra. O efeito dessa regra é executar R_0 em um estado em que a variável v está associada a um valor importado de um universo especial chamado *Reserve*. Este universo está contido em X – o superuniverso dos estados da máquina – e contém todos os elementos que serão importados.

Em geral, regras *import* são utilizadas para "estender"universos, isto é, adicionar novos elementos aos universos. Assim, regras da forma

$$\textbf{import } v \textbf{ do } U(v) := \mathit{true}, R_0 \textbf{ end}$$

onde U é um nome de universo e R_0 é uma regra, podem ser escritas utilizando a seguinte regra *extend*:

$$\textbf{extend } U \textbf{ with } v \textbf{ do } R_0 \textbf{ end}.$$

Regra Choose

A regra *choose* tem a forma

$$\textbf{choose } v \textbf{ in } U \textbf{ satisfying } g \textbf{ do } R_0 \textbf{ end}$$

onde v é uma variável, U é o nome de um universo finito, g é uma expressão booleana e R_0 é uma regra. O efeito desta regra é executar a regra R_0 em um estado no qual a variável v está associada a um valor pertencente ao universo U. Este valor é escolhido de maneira não-determinista e satisfaz a guarda g.

[2]Variáveis são símbolos que podem denotar elementos do superuniverso.

Regra Forall

A regra *var* tem a forma

forall v **in** U **do** R_0 **end,**

onde v é uma variável, U é o nome de um universo finito e R_0 é uma regra. O efeito desta regra é criar uma instância de R_0 para cada elemento pertencente ao universo U. Em cada instância de R_0, a variável v está associada ao elemento correspondente de U. Após criadas as instâncias, todas são executadas em paralelo.

ASM possui ainda recursos para expressar paralelismo assíncrono, como se vê na Seção 1.4. Além disso, para simplificar as especificações, supõe-se que conjuntos como o dos números inteiros, dos racionais, das listas e das partes de conjuntos estejam contidos no superuniverso de um estado.

1.2.1 Exemplos

Nesta seção, apresentam-se alguns exemplos de definição de sistemas em ASM.

Exemplo 1.2.2 (Pesquisa Binária) Este exemplo tem por finalidade mostrar a utilização de funções e regras básicas na especificação de algoritmos simples.

O problema consiste em encontrar um valor k em um arranjo ordenado de inteiros, a, de comprimento n, indexado de 1 a n. Um algoritmo bastante utilizado para resolução deste problema é a *busca binária*, que, a cada passo, restringe pela metade o espaço de busca da chave.

Na solução proposta, o arranjo a será representado pela função f, de modo que $f(k) = a[k]$, para k inteiro.

No estado inicial, *inf* é interpretado como 1, *sup* é interpretado como n, k é interpretado como o valor da chave de pesquisa, o nome de relação *encontrado* é interpretado como *false* e o nome de função f é interpretado como uma função que, aplicada a um número inteiro i, $1 \leq i \leq n$, retorna $a[k]$. Todos os outros nomes de função são interpretados como *undef*.

A regra da especificação é

if (**not** *encontrado* **and** $inf \leq sup$) **then**
  if $(k = f((inf + sup)/2))$ **then**
    $encontrado := true$,
    $pos := (inf + sup)/2$
  elseif $(k < f((inf + sup)/2))$ **then**
    $sup := (inf + sup)/2 - 1$
  else

```
            inf := (inf + sup)/2 + 1
        end
    end
```

A máquina executa diversos passos até que a chave seja encontrada ou o algoritmo seja capaz de determinar que a chave não está presente no arranjo. Se a chave for encontrada, então $encontrado = true$ e o nome de função pos é interpretado como a posição no arranjo onde está a chave, isto é, $f(pos) = a[pos] = k$.

□

Exemplo 1.2.3 (Ordenação por seleção) Da mesma forma que no Exemplo 1.2.2, a função f representará os elementos que se deseja ordenar. Quando a execução convergir, espera-se que se $1 \leq i \leq j \leq n$, então $f(i) \leq f(j)$.

No estado inicial, o nome de função $modo$ é interpretado como 1, i é interpretado como 1 e f é interpretado como no Exemplo 1.2.2. Todos os outros nomes de função são interpretados como *undef*.

A regra da especificação é a seguinte:

```
if modo = 1 and i < n then
    k := i, j := i + 1, modo := 2
elseif modo = 2 then
    if j > n then
        modo := 3
    elseif f(j) < f(k) then
        k := j
    end,
    j := j + 1
elseif modo = 3 then
    if k ≠ i then
        f(k) := f(i), f(i) := f(k)
    end,
    i := i + 1, modo := 1
end
```

A máquina executa até que i atinja o valor n. Neste estado, o arranjo está ordenado.

□

Exemplo 1.2.4 (Números Primos) Neste exemplo, especifica-se um algoritmo bastante simples para encontrar todos os primos menores ou iguais a um número n qualquer. Para

isso, define-seum universo *Numeros*, subconjunto dos números inteiros, tal que $Numeros = \{x \in Inteiros : 2 \leq x \leq n\}$.

O vocabulário contém os nomes de função *primo* de aridade 1 e x e n de aridade zero. No estado inicial, faz-se x ser interpretado como o número 3 e a função *primo* aplicada a qualquer elemento pertencente ao universo *Numeros* retornar *true*. A regra marcará com false todos os números de 2 a n que não forem primos.

A regra da especificação é a seguinte:

```
if x ≤ n then
    forall y in Numeros do
        if y < x and x%y = 0 then
            primo(x) := false
        end
    end,
    x := x + 1
end
```

A máquina executará vários passos até que x atinja o valor $n + 1$. Neste estado, o nome de função *primos* será interpretada como uma função que, aplicada a um número inteiro k, $2 \leq k \leq n$, retorna *true*, se k for um número primo, ou *false*, se k for um número composto.

□

1.3 O Modelo Formal de ASM

Nesta seção apresenta-se o modelo formal de ASM, conforme definido no *Lipari Guide* [23], e mostra-se um exemplo bastante simples de definição ASM e uma prova de correção da especificação apresentada. Um tutorial para a metodologia pode ser encontrado em [36].

1.3.1 Definição da Máquina

ASM são sistemas de transição que especificam computação cujos estados são álgebras [17]. Os universos de álgebras que formam os estados da computação constituem o superuniverso da ASM.

Definição 1.3.1 (Vocabulário) Um *vocabulário* Υ é uma coleção finita de nomes de função e relação, cada nome com uma aridade fixa. Estão presentes em todo vocabulário: o sinal de igualdade, *true*, *false* e *undef* e os operadores booleanas usuais. □

A função *undef* é utilizada para modelar funções parciais. Inicialmente, toda função possui este valor para todos os seus pontos.

Definição 1.3.2 (Estado) Um *estado* S é uma álgebra[3], dada por:

- um vocabulário Υ, o vocabulário do estado S;
- um conjunto X, denominado o superuniverso de S, no qual estão contidos os conjuntos dos números inteiros, dos valores lógicos, dos números racionais, das cadeias de caracteres, das listas, das tuplas e das partes de conjuntos;
- uma função $Val : \Upsilon \rightarrow (X^* \rightarrow X)$, que fornece a interpretação dos nomes de funções pertencentes a Υ em funções de $X^* \rightarrow X$.

Para os conjuntos do superuniverso, estão definidas as operações usuais – por exemplo, para os inteiros, estão definidas as operações de adição, multiplicação, etc. □

A noção de estado é de extrema importância no contexto de ASM, pois cada estado representa um passo na execução da máquina abstrata.

Definição 1.3.3 (Funções Estáticas e Funções Dinâmicas) Uma função é *dinâmica* se puder sofrer atualizações, isto é, se durante uma mudança de estado, a sua interpretação puder ser modificada em algum ponto. Caso contrário, diz-se que a função é *estática*. □

Os conceitos de funções estáticas e dinâmicas são importantes nas definições de regras e atualização, dadas abaixo. Basicamente, o caráter dinâmico do sistema é modelado pelas alterações, de estado para estado, na interpretação de funções dinâmicas.

Definição 1.3.4 (Termo) *Termos* são definidos recursivamente como:

- uma variável é um termo;
- um nome de função ou relação de zero argumento é um termo;
- se f é um nome de função ou relação r-ária, $r > 0$, e $\bar{x} = (x_1, \ldots, x_r)$ é uma tupla de r elementos, então $f(\bar{x})$ é um termo.

□

Termos sem variáveis são interpretados, em um estado S, da seguinte maneira:

[3]Uma definição de *Álgebra* pode ser encontrada em [18]

- se f é um nome de função ou relação de zero argumento, sua interpretação é $Val_S(f)$;
- se f é um nome de função r-ária e $(x_1, \ldots, x_r)$ é uma tupla de comprimento r, então

$$Val_S(f(x_1, \ldots, x_r)) = Val_S(f)(Val_S(x_1), \ldots, Val_S(x_r))$$

Definição 1.3.5 (Endereço) Um *endereço* em um estado $S = (\Upsilon, X, Val)$ é um par $(f, \bar{x})$, onde $f \in \Upsilon$ é um nome de função dinâmica, e $\bar{x} \in X^*$ é uma tupla cujo comprimento é igual à aridade de f. □

Definição 1.3.6 (Atualização) Uma *atualização* em um estado $S = (\Upsilon, X, Val)$ é um par (l, y), onde l é um endereço em S e $y \in X$. □

A noção de atualização é importante para a definição de regras e disparo de regras. Para cada regra define-se um conjunto de atualizações, que conterá os pontos em que haverá mudança na interpretação de alguns nomes de funções dinâmicas no estado seguinte.

Definição 1.3.7 (Consistência de um Conjunto de Atualizações) O conjunto de atualizações Δ é *consistente*, se

$$[((f, \bar{x}), y_1) \in \Delta \wedge ((f, \bar{x}), y_2) \in \Delta] \Rightarrow [y_1 = y_2,]$$

isto é, se não houver, em Δ, duas atualizações diferentes para o mesmo endereço. □

Definição 1.3.8 (Regra) *Regras* são definidas recursivamente por:

- a regra de atualização $R \equiv f(\bar{t}) := t_0$ é uma regra; nesta expressão, f é um nome de função dinâmica, e t_0 e $\bar{t}$ são termos; se f é um nome de relação, então t_0 deve ser booleano.

  Define-se o conjunto de atualizações de R em um estado S, $Updates(R, S)$, como o conjunto $\{(l, y)\}$, onde $l = (f, Val_S(\bar{t}))$ e $y = Val_S(t_0)$;

- um bloco de regras $R \equiv R_1, \ldots, R_k$ é uma regra; o seu conjunto de atualizações é dado por

$$Updates\,(R, S) = \bigcup_{i=1}^{k} Updates\,(R_i, S);$$

  isto significa que, para disparar um bloco de regras, disparam-se todas as regras que o compõem *simultaneamente*. Note que a ordem de $R_1, \ldots, R_k$ não é relevante na execução do bloco;

- se k é natural, $g_0, \ldots, g_k$ são termos booleanos e $R_0, \ldots, R_k$ são regras, então

$$R \equiv \textbf{ if } g_0 \textbf{ then } R_0 \textbf{ elseif } g_1 \textbf{ then } R_1 \ldots \textbf{ elseif } g_k \textbf{ then } R_k \textbf{ endif}$$

  é uma regra. O conjunto de atualizações desta regra é dado por

$$\mathit{Updates}(R, S) = \begin{cases} \mathit{Updates}(R_i, S) & \text{se } g_i \wedge \forall j < i : \neg g_j \\ \emptyset & \text{se } \forall i : \neg g_i \end{cases}$$

Estas regras são conhecidas como *regras básicas.* □

Estas regras são as mais simples de ASM e permitem a especificação de apenas alguns tipos de sistemas reais. Outros tipos de regras mais poderosas serão mostrados nas seções subseqüentes. Observe que **não existe** composição seqüencial de regras, isto é, não há uma regra da forma

$$R_1; R_2$$

onde primeiro executa-se R_1 e em seguida R_2. Isto porque um programa em ASM descreve somente um passo do algoritmo. A composição seqüencial pode gerar estruturas de execução muito complexas, o que pode dificultar o raciocínio sobre a especificação [24].

Definição 1.3.9 (Disparo de uma Regra) O *disparo de uma regra* R em um estado S produz um novo estado S', tal que, se $\mathit{Updates}(R, S)$ for consistente, então

$$\mathit{Val}_{S'}(f, \bar{x}) = \begin{cases} y & \text{se } ((f, \bar{x}), y) \in \mathit{Updates}(R, S) \\ \mathit{Val}_S(f, \bar{x}) & \text{caso contrário.} \end{cases}$$

Se o conjunto $\mathit{Updates}(R, S)$ não for consistente, então o efeito do disparo de R em S será nulo, isto é, o estado S' resultante será igual a S. Outra abordagem utilizada para tratar conjuntos de atualização inconsistentes é fazer uma escolha não-determinista da atualização que será disparada [29].

Define-se $\mathit{Fire}(R, S)$ como o estado resultado do disparo de R em S.

□

Definição 1.3.10 (Especificação ASM) Uma *especificação ASM* é uma quádrupla da forma $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$, onde

- $\Upsilon = \Upsilon_0 \cup \Upsilon_e$ é um vocabulário composto pela união de um vocabulário pré-definido, Υ_0, e um vocabulário especificado pelo usuário, Υ_e.

  O vocabulário pré-definido Υ_0 contém os nomes de relação *Boolean*, *Integer*, *String*, *List*, *Set*, que serão interpretados, respectivamente, como os universos dos valores lógicos, dos números inteiros, das cadeias de caracteres, das listas e das partes de conjuntos. Υ_0 contém também as operações usuais sobre estes conjuntos;

- $\mathcal{A}$ é um conjunto de Υ-álgebras ou estados S, cada um consistindo no vocabulário Υ e um conjunto X (o superuniverso de S) de funções, que são as interpretações em X dos nomes de funções de zero argumento pertencentes a Υ. Um nome de função de aridade r, $r > 0$, é interpretado como uma função de $X^r \to X$. O conjunto X é comum a todos os estados pertencentes a $\mathcal{A}$;
- $S_0 \in \mathcal{A}$ é o estado inicial, onde a interpretação de alguns nomes de funções são dados; nomes de funções cujas interpretações não são dadas em S_0 são interpretados como *undef*;
- $\mathcal{P}$ é uma regra que descreve as modificações das interpretações de nomes de funções de uma álgebra (estado) para outra.

□

O vocabulário de uma ASM reflete apenas os recursos verdadeiramente invariantes de um algoritmo, em vez de detalhes de um estado em particular.

O estado inicial S_0 pode ser definido através de qualquer formalismo existente, em particular, via regras de transição do modelo ASM. Pode-se utilizar também mecanismos como lambda cálculo.

Definição 1.3.11 (Execução de uma Especificação ASM) A *execução de uma especificação ASM* $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$ é uma seqüência $\mathcal{S} = \langle S_n : n \geq 0 \rangle$ de estados pertencentes a $\mathcal{A}$, onde $S_{n+1} = \mathit{Fire}(\mathcal{P}, S_n)$, isto é, S_{n+1} é obtido a partir de S_n, disparando a regra $\mathcal{P}$ em S_n. □

Na maioria dos sistemas dinâmicos discretos, a execução pode ser influenciada pelo ambiente. Intuitivamente, um ambiente ativo é um agente externo. Desta maneira, utilizam-se *funções externas* na modelagem do ambiente no qual o sistema está inserido. Além disso, utilizam-se funções externas para:

- prover ocultamento de informações – qualquer característica do sistema, cujo funcionamento não é relevante no entendimento da especificação, pode ser modelada por meio de funções externas;
- inserir não-determinismo ao modelo – considerando que as ações do ambiente acontecem de maneira não-determinista, utilizam-se funções externas para descrever implicitamente o não-determinismo.

1.3.2 Exemplo de Especificação ASM

Nesta seção mostra-se um exemplo simples de especificação formal em ASM. Após apresentada a especificação, mostra-se a sua correção em relação ao mundo real.

Exemplo 1.3.1 (Função Fatorial) Um exemplo de especificação ASM para a função fatorial é $ASM_{fat} = (\Upsilon, \mathcal{A}, S_0, P_{fat})$, onde:

- $\Upsilon = \Upsilon_0 \cup \Upsilon_e$, onde Υ_0 é o conjunto dos símbolos pré-definidos e $\Upsilon_e = \{fat, i\}$;
- $\mathcal{A}$ é um conjunto de estados;
- a interpretação dos nomes de função pertencentes a Υ em $S_0 \in \mathcal{A}$ é dado pela função Val_{S_0}, tal que
$$\begin{aligned} Val_{S_0}(i) &= 0 \\ Val_{S_0}(fat) &= \lambda n.n = 0 \rightarrow 1, undef \end{aligned}$$
- P_{fat} é dado pela regra:

  $fat(i+1) := (i+1) \times fat(i)$,
  $i := i + 1$

A execução de ASM_{fat} é a seqüência $\mathcal{S}_{fat} = \langle S_0, S_1, \cdots \rangle$ de elementos de $\mathcal{A}$, onde cada $S_n = Fire(P_{fat}, S_{n-1})$, para $n > 0$. □

Proposição 1.3.1 (Correção do Exemplo 1.3.1) Para todo $i \geq 0$, $fat(i) = i!$ no estado S_i.

A demonstração por indução em i é a seguinte:

- para $i = 0$, $Val_{S_i}(i) = Val_{S_0}(i) = 0$ e
$$\begin{aligned} Val_{S_i}(fat)(Val_{S_i}(i)) &= Val_{S_0}(fat)(0) \\ &= (\lambda n.n = 0 \rightarrow 1, undef)(0) = 1 = 0! = (Val_{S_0}(i))! \\ &= (Val_{S_i}(i))! \end{aligned}$$
- supondo que, no estado S_i, $Val_{S_i}(fat)(Val_{S_i}(i)) = (Val_{S_i}(i))!$, tem-se que, no estado S_{i+1},
$$Val_{S_{i+1}}(i) = Val_{S_i}(i) + 1$$
  e
$$\begin{aligned} Val_{S_{i+1}}(fat)(Val_{S_{i+1}}(i)) &= (Val_{S_i}(i) + 1) \times Val_{S_i}(fat)(Val_{S_i}(i)) \\ &= (Val_{S_i}(i) + 1) \times (Val_{S_i}(i))! \\ &= (Val_{S_i}(i) + 1)! \\ &= (Val_{S_{i+1}}(i))! \end{aligned}$$

C.Q.D.

1.3.3 Regras Não-Básicas

Nesta seção apresentam-se as regras de transição não-básicas, que são as regras que utilizam variáveis. A introdução de variáveis ao modelo ASM dá maior poder às definições, permitindo, por exemplo, importar elementos, o que permite estender universos, e especificar o não-determinismo de sistemas.

Para a regra *import* definida a seguir, considera-se que existe um universo de nome *Reserve*, contido no superuniverso do estado. Este universo contém os elementos que serão importados. Cada regra *import* retira um elemento de *Reserve*.

Definição 1.3.12 (Regra *Import*) A regra *import* é uma regra da forma

$$R \equiv \textbf{import } v \textbf{ do } R_0 \textbf{ end},$$

onde v é uma variável e R_0 é uma regra.

A semântica desta regra é a seguinte: escolhe-se um elemento a do universo *Reserve* e associa-o à variável v. Desta forma, executa-se a regra R_0, no estado S_a, que é uma expansão do estado S, no qual interpreta-se v como a. O conjunto de atualizações para esta regra é dado por

$$Updates(R, S) = \{((Reserve, a), false)\} \cup Updates(R_0, S_a).$$

□

O efeito desta regra é "criar"um novo elemento. Em geral é utilizada com uma regra da forma $U(v) := true$, onde U é um nome de universo, para inserir o novo elemento ao universo U, modelando uma expansão de U.

Definição 1.3.13 (Regra *Forall*) A regra *var* é uma regra da forma

$$R \equiv \textbf{forall } v \textbf{ in } U \textbf{ do } R_0 \textbf{ end},$$

onde v é uma variável, U é o nome de um universo finito e R_0 é uma regra.

A semântica desta regra é a seguinte: para cada elemento x de U, executamos a regra R_0 no estado S_x, que é uma expansão do estado S, tal que a variável v é interpretada como x. Esta execução cria o conjunto de atualizações $Updates(R_0, S_x)$. O efeito da regra é a união de todos os conjuntos $Updates(R_0, S_x)$, tal que $x \in U$, isto é,

$$Updates(R, S) = \bigcup_{x \in U} Updates(R_0, S_x).$$

□

O efeito desta regra é criar uma instância de R_0 para cada elemento pertencente ao universo U. Em cada instância de R_0, a variável v está associada ao elemento correspondente de U. Após criadas as instâncias, todas são executadas em paralelo. Esta regra é usada para modelar paralelismo síncrono, como está mostrado no Exemplo 1.2.4.

Definição 1.3.14 (Regra *Choose*) A regra *choose* é uma regra da forma

choose v **in** U **satisfying** g **do** R_0 **end**,

onde v é uma variável, U é o nome de um universo finito, g é um termo booleano, e R_0 é uma regra. O efeito desta regra é executar a regra R_0 em um estado S_a, no qual a variável v está associada a um elemento $a \in U$. O elemento a é escolhido de maneira não-determinista e torna a guarda g verdadeira.

□

Algoritmos não-deterministas são úteis, por exemplo, como descrições (especificações) em alto nível de algoritmos "reais". Esta regra é usada para modelar explicitamente o não-determinismo. Outra maneira de modelar o não-determinismo é permitir que a inconsistência de conjuntos de atualização sejam resolvidas escolhendo-se não-deterministicamente qual atualização disparar. Entretanto, esta abordagem pode dificultar a leitura de uma especificação e a demonstração de propriedades sobre ela. Não-determinismo pode ser modelado ainda por meio de funções externas, deixando-se a escolha por conta do ambiente.

1.4 ASM Multiagentes

Uma *ASM Multiagente* $\mathcal{M}$, também conhecida como *ASM Distribuída*, contém um número finito *agentes* computacionais que executam concorrentemente um número finito de programas. A cada agente está associado um programa. Formalmente, uma ASM Multiagente $\mathcal{M}$ é uma tupla $(\Upsilon_{\mathcal{M}}, \mathcal{A}, \mathcal{C}_0, \mathcal{P})$, tal que:

- $\Upsilon_{\mathcal{M}}$ é um vocabulário que contém os nomes de função que aparecem nas regras de $\mathcal{P}$, com exceção de *Self*. Pertencem a $\Upsilon_{\mathcal{M}}$ os nomes de funções de zero argumento que representam os nomes de módulos (elementos de $\mathcal{P}$) e uma função unária, *Mod*, que representa a relação entre agentes e módulos. Um elemento a é um *agente* em um dado estado S se houver um nome de módulo μ tal que $S \models \mathit{Mod}(a) = \mu$. Da mesma forma, o programa de a é $\mathit{Prog}(a) = \pi_\mu$.
- $\mathcal{A}$ é um conjunto de estados;
- $\mathcal{C}_0$ é uma coleção de estados denominados *estados iniciais* de $\mathcal{M}$; todos os elementos de $\mathcal{C}_0$ compartilham as mesmas interpretações dos nomes de função de $\Upsilon_{\mathcal{M}}$, com exceção de *Self*, que é interpretado como o agente associado ao estado;

- $\mathcal{P}$ é um conjunto finito indexado de programas π_μ (os módulos), identificados pelos nomes de módulos $\mu \in \Upsilon$; cada programa pertencente a $\mathcal{P}$ é uma regra.

O nome de função especial de zero argumento *Self* permite a auto-identificação de agentes: *Self* é interpretado como a por cada agente a e pode ser usada, por exemplo, para modelar algum estado local dos agentes. Diz-se que um agente a realiza uma *transição* (*"move"*) em S se o conjunto de atualizações

$$Updates(a, S) = Updates(Prog(a), View_a(S))$$

é disparado em S, isto é, se a regra que forma o programa do agente a for disparada no estado $View_a(S)$, que é a expansão do estado S, em que a variável *Self* é interpretada como a.

Sobre esta definição de transição, pode-se definir diversos tipos de execução. Abaixo, tem-se a definição de *execução parcialmente ordenada*, que é uma importante noção de computação distribuída. Esta definição garante que não haverá inconsistências na execução de 2 ou mais agentes, pois cada transição é executada por um agente de maneira atômica.

Definição 1.4.1 (Execução parcialmente ordenada) Uma execução parcialmente ordenada ρ de uma ASM Multiagentes $\mathcal{M}$ é uma tripla (M, A, σ) que satisfaz as seguintes condições:

1. M é um conjunto parcialmente ordenado, no qual todos os conjuntos de transição $\{y : y \preceq x\}, x \in M$, são finitos: os elementos de M representam transições realizadas pelos vários agentes durante a execução[4].
2. A é uma função sobre M tal que todo conjunto não-vazio $\{x : A(x) = a\}$ é linearmente ordenado: $A(x)$ é o agente que realiza a transição x (supõe-se que as transições de um único agente sejam linearmente ordenados).
3. σ é uma função que associa um estado de $\mathcal{M}$ a cada segmento inicial[5] de M, tal que $\sigma(\emptyset)$ é um estado inicial: para cada segmento inicial X de M, $\sigma(X)$ é o estado que resulta da realização de todos os movimentos em X.
4. A *condição de coerência*: se x é um elemento maximal em um segmento inicial finito X de M e $Y = X - \{x\}$, então $A(x)$ é um agente em $\sigma(Y)$ e $\sigma(X)$ é obtido a partir de $\sigma(Y)$ disparando $A(x)$ em $\sigma(Y)$.

□

Nos exemplos a seguir, utiliza-se esta noção de execução.

[4] Duas transições x e y são tais que $y \prec x$, se a transição x iniciou após o término de y

[5] Um *segmento inicial* de um conjunto parcialmente ordenado P é uma sub-estrutura X de P tal que

$$x \in X \wedge y < x \Rightarrow y \in X.$$

1.4.1 Exemplos de ASM Multiagente

Nesta seção apresentam-se exemplos de utilização de ASM multiagentes para definição de sistemas distribuídos. Os sistemas especificados são o problema do *Jantar dos Filósofos* e o protocolo *Alternating Bit*.

Exemplo 1.4.1 (Jantar dos Filósofos) Um conjunto de *filósofos* senta-se a uma mesa circular, e um garfo é posicionado entre cada dois filósofos. Cada um alterna seu estado entre pensando ("*thinking*"), com fome ("*hungry*") e comendo ("*eating*"). Para entrar no estado "eating", um filósofo deve estar com os dois garfos adjacentes a ele em seu poder. Cada garfo só pode estar servindo a um filósofo de cada vez.

O código apresentado a seguir corresponde ao programa de um agente ASM. Cada agente representa um filósofo distinto e nenhuma suposição é estabelecida sobre a velocidade relativa da execução do código de cada um. Suponha que p seja uma função equivalente a *Self* citada anteriormente, associada a instâncias do universo *Philosophers*. A função *Status*(p) representa o estado atual do filósofo p, a função estática *Next*(p) representa o filósofo imediatamente à esquerda de p, e *Holder*(f) indica qual é o filósofo que, num dado momento, possui o garfo f. Observe que, inicialmente, *Holder*(f) = *undef*, para todo f, indicando que os garfos não estão sendo usados.

if *Status*(p) = "*thinking*" **then**
    Status(p) := "*hungry*"
end,
if *Status*(p) = "*hungry*" **and** *Holder*(*LeftFork*(p)) = *undef*
    and *Holder*(*RightFork*(p)) = *undef* **then**
    Holder(*LeftFork*(p)) := p**,**
    Holder(*RightFork*(p)) := p**,**
    Status(p) := "*eating*"
end,
if *Status*(p) = "*eating*" **then**
    Holder(*LeftFork*(p)) := *undef***,**
    Holder(*RightFork*(p)) := *undef***,**
    Status(p) := "*thinking*"
end

A solução apresentada é isenta de *deadlock*, uma vez que os dois garfos são requisitados atomicamente. A inicialização dos valores das funções é apresentada a seguir. Observe que todas as atualizações são executadas em paralelo, diferentemente do código acima, onde cada instância representa um agente diferente.

forall q **in** *Philosophers*
    Status(q) := "*thinking*"**,**

$Holder(LeftFork(q)) := undef$
$Holder(RightFork(q)) := undef$
end

□

Exemplo 1.4.2 (*Alternating Bit Protocol*) Um agente é designado por *Sender* e envia mensagens a outro agente, designado por *Receiver*, através de um meio onde as mensagens podem ser perdidas ou duplicadas, mas a ordem de envio é preservada. Para identificar as mensagens corretas, um bit adicional é enviado, alternando 0 e 1 a cada transmissão.

Uma especificação do código dos agentes *Sender* e *Receiver* é apresentada a seguir.

Agente *Sender*:

if $Clock > Slast + timeout$ **then**
$SRmessage := Smessage$,
$SRbit := Sbit$,
$Slast := Clock$
end,
if $RSbit = Sbit$ **then**
$Scont := Scont + 1$,
$Smessage := InputFile(Scont + 1)$,
$Sbit := \neg Sbit$
end

Agente *Receiver*:

if $Clock > Rlast + timeout$ **then**
$RSbit := Rbit$,
$Rlast := Clock$
end,
if $SRmessage \neq undef$ **and** $SRbit \neq Rbit$ **then**
$Rcont := Rcont + 1$,
$OutputFile(Rcont + 1) := SRmessage$,
$Rbit := \neg Rbit$
end

A função externa *Clock* é utilizada para representar tempo transcorrido. O envio de mensagens pode ser repetido após um intervalo determinado pela função estática *timeout*. É razoável supor que as duas referências à função *Clock*, em cada agente, retornem o mesmo valor em

um mesmo passo de execução. As mensagens são geradas em *Sender* pela função *InputFile* e armazenadas em *Receiver*, na função *OutputFile*.

As funções *Smessage* e *Sbit* armazenam o valor da mensagem e do bit atuais em *Sender*, enquanto *Rbit* é o valor do bit em *Receiver*. As funções *SRmessage* e *SRbit* representam os dados em trânsito de *Sender* para *Receiver*, enquanto *RSbit* representa o bit enviado pelo canal de comunicação na direção inversa. A função $\neg$ é utilizada para gerar os valores alternados para os bits.

Além dos dois agentes apresentados acima, a especificação conta ainda com os agentes *SRLose* e *RSLose*, que simulam a perda ocasional de mensagens nos canais de transmissão.

Agente *SRLose*:

$SRbit := undef$,
$SRmessage := undef$

Agente *RSLose*:

$RSbit := undef$

Esses agentes eliminam as informações armazenadas nos "canais" de comunicação. Como nenhuma suposição é estabelecida sobre a velocidade relativa de execução dos agentes, a perda de mensagens acontece aleatoriamente.

Para o correto funcionamento do protocolo, as funções *Sbit*, *Rbit*, *Scont*, *Rcont*, *Slast* e *Rlast* devem ser todas inicializadas com o valor zero. □

1.5 Conclusões

Uma das grandes vantagens do modelo é a possibilidade de se executar uma especificação, o que pode facilitar a tarefa de encontrar erros. O modelo possui também recursos para modelar concorrência e não-determinismo, como pode ser visto no exemplo do Jantar dos Filósofos na Seção 1.4.1.

Além disso, pode-se citar também a existência de uma teoria matemática subjacente, a teoria de Álgebras Evolutivas, que permite a prova de propriedades da especificação. O modelo de Linguagens imperativas não possuem tal característica. Outros modelos que possuem base matemática formal são os modelos Funcional, Lógico e Algébrico, mas estes não permitem execução direta na Máquina de von Neumman.

Capítulo 2

A Linguagem Machĭna

Neste capítulo, são apresentadas a sintaxe e semântica da linguagem Machĭna, projetada para implementar especificações formais baseada no modelo ASM.

2.1 Introdução

A linguagem Machĭna é baseada no conceito de Máquinas de Estado Abstratas (ASM) e possui suporte à modularidade, criação de novos tipos e construções de alto nível. Um programa consiste na definição de um vocabulário, do estado inicial e da regra de transição que promove mudança de estados. Além disso, há também na linguagem construções para definição de invariantes de execução, controle de visibilidade intermódulos e comunicação entre agentes.

A linguagem Machĭna possui como características principais:

- estruturas para modularização e mecanismos de visibilidade e proteção;
- extensibilidade de tipos;
- sequenciadores de regras;
- sistema fortemente tipado, com um rico conjunto de tipos pré-definidos;
- invariantes para a execução da regra de transição da máquina abstrata;
- regras de transição de estado;
- multiagentes, com capacidade de autonomia, independência, consciência de contexto e sociabilidade, introduzida na linguagem de maneira simples e direta;
- abstração de regras de transição, incluindo ações e iterações, que podem, por exemplo, ser executadas a partir de outras máquinas, criando a noção de submáquinas.

2.2 Elementos Básicos

2.2.1 Comentários

Comentários em Machĭna podem ser de dois tipos: (i)um texto que se inicia com `/*` e termina com `*/`, sem aninhamento; (ii) um texto que se inicia com `//` e vai até o fim da linha.

2.2.2 Literais

Os literais ou constantes de Machĭna são:
literal ::= boolean-literal | char-literal | integer-literal | float-literal
| string-literal | list-literal | agent-literal

onde:

- boolean-literal: são as constantes **false** e **true**.
- char-literal: são os caracteres do conjunto ASCII, representados por um símbolo entre apóstrofos, como $'$`5`$'$ e $'$`a`$'$, ou por um código denotando caracteres especiais, como: $'\backslash n'$ (newline), $'\backslash ddd'$ (número decimal), $'\backslash a'$ (bell), $'\backslash b'$ (backspace) , $'\backslash f'$ (formfeed), $'\backslash r'$ (carriage return), $'\backslash t'$ (horizontal tab), $'\backslash v'$ (vertical tab), $'\backslash\backslash'$(backslash), $'\backslash''$(apostrofe), $'\backslash"'$ (quote).
- integer-literal: são os números inteiros, escritos na base decimal (seqüência de dígitos), octal (`0` seguido de seqüência de dígitos de `0` a `7`) ou em hexadecimal (`0x` seguido de seqüência de dígitos ou letras de `A` a `F` ou `a` a `f`). Os números inteiros estão no intervalo $[-2.147.483.648, 2.147.483.647]$.
- float-literal: são os números de ponto flutuante, formados por seqüência de dígitos, seguida por uma parte fracionária opcional, seguida por uma parte de expoente opcional. A parte fracionária consiste em um ponto (`"."`) seguido por uma seqüência de dígitos. A parte do expoente consiste em `e` ou `E`, seguido por um sinal opcional (`+` ou `-`), seguido por uma seqüência de dígitos. Números de ponto flutuante têm valor máximo $\pm$1.79769313486231570E+308 e valor mínimo $\pm$4.94065645841246544E-324, seguindo o padrão IEEE 754 para ponto flutuante de 64 bits.
- string-literal: são as cadeias de caracteres, que são representadas por seqüência de caracteres entre aspas, por exemplo, `"string"`, `"isto e\' um teste"`, `"fim de linha\n"`, etc. O tipo de um string-literal é o funcional **Int -> Char**. Um string-literal representa a definição desta função para valores de seu domínio a partir de 1. Para valores de domínio superior ao comprimento da cadeia, ou para valores menores que 1, a imagem da função é **undef**.
- list-literal: é a constante **nil**, que representa a lista de qualquer tipo vazia.

- agent-literal: é a constante **self** do tipo **agent of** `M`, sendo `M` o módulo do agente. A constante **self** têm a identificação do agente que consulta seu valor.

2.2.3 Identificadores

Um identificador em Machĭna é formado por uma sequência de qualquer tamanho de letras e dígitos, necessariamente começando por uma letra. O caractere *underscore* conta como letra. Na formação de identificadores, letras minúsculas, maíusculas ou acentuadas são todas distintas. O tamanho de um identificador é ilimitado, entretanto o comprimento de identificadores de funções externas têm as limitações do ambiente de execução.

Identificadores são representados pelo sintagma `identifier` e podem denotar palavras-chave ou os seguintes elementos de uma especificação Machĭna: `machina-name`, `module-name`, `interface-name`, `agent-name`, `rule-name`, `type-name`, `function-name`, `variable-name`, `parameter-name` e `constant-name`.

2.2.4 Palavras Reservadas

Os seguintes identificadores são declarados como palavras reservadas de Machĭna:

abstractions	**action**	**agent**	**Agent**	**algebra**	**all**
and	**as**	**begin**	**blocked**	**Bool**	**case**
Char	**choose**	**create**	**dead**	**default**	**derived**
destroy	**dispatch**	**do**	**dynamic**	**else**	**elseif**
end	**ensure**	**enum**	**exist**	**external**	**false**
file	**forall**	**if**	**import**	**include**	**in**
initial	**inout**	**input**	**Int**	**interface**	**invariant**
is	**let**	**list**	**List**	**loop**	**machina**
new	**module**	**not**	**of**	**or**	**otherwise**
out	**Output**	**public**	**Real**	**return**	**require**
rule	**running**	**satisfying**	**select**	**self**	**set**
Set	**shared**	**state**	**State**	**static**	**step**
stop	**String**	**then**	**true**	**tuple**	**type**
unborn	**undef**	**with**	**xor**		

2.3 Notação Para Sintaxe

Para descrição da gramática, utiliza-se a BNF estendida [39], ou XBNF. As construções utilizadas são semelhantes à BNF tradicional, com as seguintes extensões:

|	Opções
{ X }	Zero ou mais ocorrências de X
[X]	Zero ou uma ocorrência de X

Os caracteres que representam meta-símbolos, se utilizados na gramática, são escritos em aspas, por exemplo "{".

2.4 Unidades de Especificação

Uma especificação formal em Machĭna consiste em um conjunto de *unidades de especificação*, as quais podem ser um módulo de definição *machina*, *módulos de programa*, ou definições de interface de agentes.

Sintaxe:
specification-unit ::= program-module | agent-interface | machina-definition

Cada unidade de especificação pode ser compilada separadamente, desde que se respeite a ordem de dependência entre os módulos de programa, estabelecida pelas diretivas **include**.

2.4.1 Módulos de Programa

Um módulo de programa especifica a regra que um agente a ela associado executa, seu vocabulário, i.e., conjunto de símbolos que manipula, interpretação destes símbolos no estado inicial e o invariante da execução. Um módulo contém um mecanismo de controle de visibilidade, permitindo organizar o vocabulário de um agente em unidades encapsuladas, reduzindo sua complexidade.

Um módulo de programa tem a forma:

module nome-do-módulo
  import elementos importados
  include elementos incluídos
  algebra:
    elementos declarados (funções e tipos)
  abstractions:
    declaração de abstrações (ações e iterações)
  initial state:
    inicializações de funções dinâmicas
  transition:
    regra de transição de estado
  invariant: invariante da execução
end nome-do-módulo

A parte **import** coloca no escopo do módulo a interface dos agentes com os quais o agente do módulo deseja se comunicar. A interface de um agente contém as assinaturas de abstrações de regras que o módulo do agente torna público para uso de outros agentes.

A parte **include** define os módulos secundários e seus elementos que devem ser incorporados ao módulo. Esta cláusula tem importante papel na formação do vocabulário dos agentes. A cláusula **include** também serve para controle de visibilidade de elementos declarados públicos ou compartilhados em um módulo.

A seção **algebra** define os elementos da álgebra subjacente ao modelo, contendo os *sorts* ou tipos e as funções do módulo.

A seção **initial state** serve para inicialização de funções dinâmicas.

A seção **abstrations** define abstrações de regras de transição, que podem ser usadas localmente ou exportadas.

A seção **transition** define a regra de transição de estado do agente, a qual é executada repetidamente quando o agente é disparado. Esta seção representa o corpo do programa dos agentes associados ao módulo.

A seção **invariant** define a condição envolvendo os elementos de um módulo que deve ser invariante durante sua execução. Entre duas iterações da regra de transição do módulo, o invariante é verificado pelo sistema de execução Machĭna. Caso seja violado, uma mensagem de erro é emitida e a execução, interrompida.

A sintaxe de um módulo de programa é a seguinte:

program-module	::=	**module** module-name module-body **end** [module-name]
module-body	::=	[declaration-part] [initial-state-part] [transition-part] [invariant-part]
declaration-part	::=	{ foreign-part } [algebra-part] [abstraction-part]
foreign-part	::=	import-part | include-part
import-part	::=	**import** importation-list [;]
include-part	::=	**include** inclusion-list [;]
algebra-part	::=	**algebra:** { algebra-section }
abstraction-part	::=	**abstractions:** { abstraction-section }
initial-state-part	::=	**initial state:** single-transition
transition-part	::=	module-transition
module-transition	::=	**transition:** main-transition
invariant-part	::=	**invariant:** expression
algebra-section	::=	type-section | function-section | external-section
abstraction-section	::=	[**public**] rule-abstraction [;]
type-section	::=	[**public**] type-declaration ;
function-section	::=	[**public**] [**shared**] function-declaration ;
external-section	::=	external-declaration ;

2.4.2 Mecanismos de Visibilidade

Um módulo pode incorporar no vocabulário de seus agentes os elementos (declarações de funções, tipos, ações e iterações) definidas em outros módulos. A primeira parte de um módulo consiste na enumeração das interfaces que ele importa e módulos a serem incluídos. Quando um módulo é incluído, todos os seus símbolos públicos tornam-se visíveis.

Para evitar conflitos com nomes declarados localmente, as referências a nomes de tipos, agentes e funções podem ou não ser qualificadas pela entidade que os declara de acordo com a sintaxe:

agent-designator	::=	qualifier agent-name
rule-designator	::=	qualifier rule-name
type-designator	::=	qualifier type-name
function-designator	::=	qualifier function-name
variable-designator	::=	variable-name
qualifier	::=	[module-name . | agent-name .] { expression . }

Toda declaração é automaticamente privada ao módulo no qual foi declarada, isto é, declaração é visível somente dentro do módulo no qual foi feita. Um elemento só não é privado a um módulo se for explicitamente declarado como público, por meio do modificador de visibilidade **public**. Elementos declarados públicos somente são visíveis por outros módulos do mesmo agente, isto é, nenhum agente tem visibilidade direta a funções de outros agentes.

Os elementos públicos de um módulo M1 tornam-se visíveis dentro de outro módulo M2 do mesmo agente, quando o vocabulário de M1 for incluído em M2 por meio da cláusula **include**, cuja lista de inclusão é definida da seguinte forma:

inclusion-list	::=	module-inclusion { , module-inclusion }
module-inclusion	::=	module-name [included-list]
included-list	::=	(included-element { , included-element })
included-element	::=	function-name | rule-name | type-name

A cláusula **include** *causa a duplicação* de declarações de todas as funções não compartilhadas dos módulos incluídos. Somente uma duplicação das funções de um módulo incluído é feita para formação de um agente, mesmo se o módulo for incluído mais de uma vez. Desta forma, os vocabulários dos agentes de um dado módulo têm os mesmos nomes e cardinalidade, mas cada agente tem sua própria interpretação destes símbolos.

Funções declaradas **shared** não são duplicadas. Essas funções pertencem ao vocabulário comum a todos os agentes, e consequentemente, não pode haver colisão de nomes de funções compartilhadas. Funções compartilhadas ocorrem na lista de inclusão de cláusula **include** apenas para fins de controle de visibilidade.

Considere o esqueleto de módulo:

```
module M2
    include M1          Torna visíveis em M2 todos os elementos públicos de M1
    include M1(a,b)     Idem, mas os elementos a e b não precisam de qualificação
    ...
end M2
```

Na primeira inclusão, todos os elementos públicos de `M1` ficam visíveis dentro de `M2`. Um elemento `a`, público de `M1` pode ser acessado em `M2` da forma `M1.a`. Na segunda inclusão, os elementos `a` e `b` de `M1` ficam diretamente visíveis em `M2`, não sendo necessária a qualificação do módulo para se ter acesso aos elementos `a` ou `b`, isto é, estes identificadores podem ser acessados diretamente em `M2`, sem a necessidade de se especificar o módulo a que pertencem. Por exemplo, considere os módulos abaixo:

```
module X              module Y              module Z
  public a:Int;         include X;            include X, Y(n);
  b:Int;                public m,n:Int;       ...  X.a ...
  ...                   ...  X.a ...          ...  n ...  Y.m
end                   end                   end
```

O módulo `X` define um elemento de nome `a`, que é visível a partir de outros módulos, enquanto a declaração de `b` é visível somente dentro de `X`, isto é, é privativa deste módulo. A linha **include** `X`, no módulo `Y`, inclui todos os elementos públicos de `X`, o que significa que o elemento `a`, em `X`, é visível dentro de `Y`, sob o nome de `X.a`. Da mesma maneira, `a` é visível no módulo `Z`, sendo necessária a qualificação do módulo, assim como no módulo `Y`. Entretanto, como a importação do módulo `Y` em `Z` explicitou quais identificadores são visíveis, podemos utilizar o nome `n` em `Z` diretamente, sem a necessidade de escrever `Y.n`. Observe que o nome `m`, definido como público em `Y` também é visível em `Z`, mas requer completa qualificação, pois não foi explicitado na importação.

Por exemplo, suponha que o módulo `X` seja definido por:

```
module X
  include Y;
  include Z(b);
  ...
end
```

Se `a` for um símbolo público de `Y`, então `a` poderá ser utilizado dentro de `X`, qualificado pelo nome do módulo, da forma `Y.a`. Desta maneira, pode haver outro símbolo de nome `a` definido dentro de `X`, sem que haja conflitos de nomes. Para evitar a qualificação do identificador pelo nome do módulo, basta listá-lo no momento da inclusão, como na inclusão de `Z(b)`. Neste caso, `b` deve ser um símbolo público de `Z` e poderá ser utilizado dentro de `X` sem a qualificação. Entretanto, ao contrário do primeiro caso, não poderá haver em `X` a definição de outro elemento de nome `b`.

A diretiva **include** permite construir um grafo dirigido no qual os nodos são módulos e as arestas definem a relação de inclusão. Este grafo fornece a informação necessária para se

identificar os módulos que formam os agentes criados a partir de qualquer módulo. Por razões pragmáticas, não se permite ciclos neste grafo. Módulos que se incluem mutuamente não são permitidos. O programador deve contornar a restrição pela fusão dos módulos mutuamente recursivos.

2.4.3 Interfaces de Agentes

Interface é um recurso para se definir um canal de comunicação através do qual um agente pode interagir com outros agentes. A interface tem o mesmo nome do módulo principal que forma o agente, e nela são relacionados os nomes dos tipos, assinaturas de regras e abstrações exportados pelo módulo correspondente, isto é, aqueles elementos que podem ser usados por outros agentes para transmitir informação ao agente associado à interface ou dele solicitar execução de serviços.

As assinaturas das abstrações especificadas na interface têm informações redundantes em relação às suas respectivas declarações no módulo correspondente. Toda abstração cuja assinatura ocorre em alguma interface deve ser declarada pública e seus parâmetros devem possuir indicação explícita de sua natureza (**in**, **out**, **inout** ou **action**), o qual é usado no processo de comunicação interagentes.

Interfaces podem ser importadas, via a cláusula **import**, por outros módulos. Diferentemente do processo de inclusão, feito via **include**, o vocabulário da interface importada **não** é duplicado no módulo importador. O mecanismo de importação apenas estabelece um canal de comunicação. As diretivas **include**são usadas para construir um agente a partir dos seus módulos. Agentes construídos a partir de um mesmo módulo, compartilham o mesmo vocabulário, mas cada um tem sua própria interpretação dos elementos deste vocabulário.

A sintaxe da interface é:

agent-interface	::=	**interface** interface-name interface-body **end** [interface-name]
interface-body	::=	interface-section { interface-section }
interface-section	::=	exported-type | exported-abstraction
exported-type	::=	**type** type-name [;]
exported-abstraction	::=	**action** rule-name [(formal-parameters)] ;
formal-parameters	::=	formal-parameter { , formal-parameter }
formal-parameter	::=	[transmission-mode] [parameter-name :] type-expression
transmission-mode	::=	**in** | **out** | **inout**

Funções compartilhadas (**shared**) podem ser manipuladas, até mesmo concorrentemente, por todos agentes, permitindo uma forma de interação direta entre agentes. Entretanto, um agente não pode ter acesso diretamente a qualquer nome de uso exclusivo de outro agente. Neste caso, a interação entre agentes deve feita via o conceito de interface, i.e., pelo uso de ações importadas. Desta forma, os agentes escolhem livremente seus interlocutores importando as interfaces necessárias.

Funções compartilhadas podem ser qualificadas pelo nome de um agente ou pelo nome do

módulo onde foram definidas. Nesta qualificação o nome de módulo equivale ao agente **self**. Por exemplo, `M.action( ...)` denota a chamada a ação `action` do agente em execução. Já agent.action(...) é a chamada a ação `action` do agente `agent`.

A sintaxe da clásula de importação de ações e tipo de outros agentes é a seguinte:

importation-list	::=	importation { , importation }
importation	::=	interface-name [interface-import]
interface-import	::=	(interface-element { , interface-element })
interface-element	::=	rule-name | type-name

Pode-se escolher entre importar seletivamente os elementos da interface de um agente, especificando-se o nome da interface e os elementos desejados, ou então importar todos os elementos indistintamente, mencionando-se apenas o nome da interface.

2.4.4 Definições de Machĭna

Associados a uma especificação, pode haver um ou mais *módulos de definição Machĭna*, os quais criam e disparam os principais agentes que executam de forma autônoma as regras de transição de estado dos módulos.

Um módulo de definição de Máquina consiste na especificação de seu nome e de diretivas para criação dos agentes e disparo de sua execução. A partir deste módulo, o sistema de execução Machĭna cria um agente especial, denominado **superagente**, que executa as diretivas especificadas no módulo Machĭna, dando início ao processo de criação e execução dos agentes.

O módulo Machĭna possui a sintaxe seguinte:

machina-definition	::=	**machina** machina-name machina-body **end** [machina-name]
machina-body	::=	{ machina-directive }
machina-directive	::=	**agent of** module-name [number-of-agents] [;]
number-of-agents	::=	(integer-literal)

A diretiva **agent of** `M` cria um agente a partir do módulo `M`. O vocabulário do agente é formado por uma cópia própria de todas as declarações contidas no módulo M e nos módulos a partir dele incluídos via a diretiva **include**. Assim, cada agente tem sua própria e exclusiva área de dados, que, portanto, não é compartilhada com qualquer outro agente. A nenhum agente é permitido o acesso direto à área de dados de qualquer outro agente. A interação entre agentes somente é possível mediante troca de mensagens com interveniência do superagente. A cada agente está associado um programa, formado por seu conjunto de módulos, e a cada módulo pode ser associado um conjunto de agentes, os quais executam a mesma regra de transição de estado. A regra de transição de um agente de `M` é aquela contida no módulo `M`, a partir do qual foi criado. Este módulo é o módulo principal do agente. As regras de transição dos módulos incluídos a partir de `M`, denominados secundários em relação a `M`, não pertencem ao agente de `M`.

Após a criação de todos os agentes especificados no módulo Machina, a execução de cada um, em paralelo, é simultânea e automaticamente iniciada, e cada um passa a executar repetidamente sua regra de transição. Cada ciclo de execução da regra de transição de estado de um agente é atômico, sem qualquer tipo de interrupção ou bloqueio. Somente entre uma execução da regra da transição e a seguinte é que agentes podem ser bloqueados, por exemplo, por falta de recursos solicitados de outros módulos.

No exemplo abaixo, o superagente do módulo `X` cria um agente que executa a regra de `A` e seis agentes independentes que executam a regra de `B`.

```
machina X
  agent of A;
  agent of B(5);
  agent of B;
end
```

Associado ao nome de qualquer módulo, por exemplo `B`, têm-se os atributos:

- `B.numberOfAgents`, que informa o número de agentes criados para `B` em uma dada execução;
- `B.theAgent`, do tipo **Int-> agent of** `B`, que permite recuperar cada um dos agentes de B. A expressão `B.theAgent` é equivalente a `B.theAgent(1)`.
- `B.agents`, do tipo **set of agent of** `B`, que denota o conjunto de agentes criados e associados ao módulo `B`.

Todo agente possui uma função especial de nome **self** de tipo **agent of** `M`, que retorna a sua identificação, onde `M` é o módulo que contém a regra que o agente executa. As informações locais a um agente podem ser qualificadas por **self** ou pelo nome do módulo que as contém. Um elemento sem qualificação é entendido como pertencente ao agente **self**.

A execução de um agente é interrompida quando este executa o comando **stop**. Sua execução também pode ser interrompida quando as guardas da regra de transição não forem verdadeiras. Se todas as guardas do programa avaliarem em falso, então a execução da regra de transição é suspensa. Uma execução interrompida de um agente pode ser retomada mais tarde nas seguintes situações: (i) alguma guarda na regra de transição contiver alguma chamada à função externa; (ii) o agente recebeu e processou alguma resposta de pedidos que na última iteração de sua regra enviou a outros agentes; (ii) o agente recebeu e processou pedidos enviados por outros agentes.

O caso de terminação envolvendo funções externas é ilustrado abaixo:

```
module B
algebra:
  external f : Int;
  g(x : Int) : Int;
  transition:
```

```
    if f = 10 then g(10) := 0
    else g(10) := 1
    end
end
```

Quando a regra mostrada no exemplo acima for executada pela primeira vez, se o valor da função externa f for 10, o valor de g no ponto 10 será atualizado para 0. Enquanto o valor de f não mudar, a regra `g(10) := 0` será executada e não causará mudança alguma no estado. Neste caso, entretanto, a execução não será abortada, pois em algum futuro estado o valor da função externa poderá ser diferente de 10.

2.5 Comunicação entre Agentes

Uma especificação **machina** multiagente pode conter um número finito de agentes computacionais, que executam concorrentemente um número finito de programas. Os agentes podem ter sido criados pelo módulo principal máquina ou por qualquer outro agente, por meio da regra **create**. Agentes criados devem ser disparados explicitamente via a regra **dispatch** para iniciar sua execução. A execução desta regra para um agente que já esteja em execução não tem efeito algum.

Os agentes em execução comunicam-se via chamadas a abstrações de regras (**action**) que são anunciadas na interface dos módulos principais dos agentes. Durante a execução de sua regra de transição, um agente a pode solicitar a execução de uma abstração de regra disponibilizada via sua interface por um outro agente b.

Este processo é entendido como o envio de uma mensagem do agente a ao agente b, solicitando a execução do serviço definido pela abstração chamada, cujos parâmetros servem para transmitir informações ao agente b ou dele recebê-las. Estas chamadas a abstrações de regras de outros agentes são **assíncronas** em relação à execução da transição corrente, isto é, o envio de mensagens não causa bloqueio do agente durante a transição corrente, porém a próxima transição somente será retomada após o recebimento das informações por ventura solicitadas pelas mensagens enviadas durante a última iteração da regra de transição.

O envio de mensagem a um agente que tenha sido criado, mas que ainda não tenha sido disparado, portanto que ainda não esteja em execução, é considerado válido. Entretanto, a resposta à mensagem somente ocorrerá após o disparo do agente. Até que isto aconteça o agente que enviou a mensagem ficará travado, não podendo executar a transição seguinte.

Todo agente possui internamente um contador, denominado **PendingAnswers** e as filas **RequestsReceived** e **AnswersReceived**. Estas estruturas ocupam o espaço de dados de cada agente, mas não há meios na linguagem Machĭna para se ter acesso direto a elas. Sua manutenção é feita automaticamente pelo **superagente**, que controla a iteratividade das regras de transição.

A fila **AnswersReceived** tem o formato de uma lista de atualização de localizações da memória local do agente. O conteúdo desta fila é relativo ao valor de retorno de parâmetros **out** ou **inout** de chamadas a abstrações enviadas a outros agentes. Esta é a fila que coleta as respostas às chamadas executadas na última iteração da regra de transição.

A fila **RequestsReceived** armazena os pedidos de execução de abstrações de regras encaminhadas ao agente. Cada elemento da fila deve informar o nome da abstração a ser executada e seus respectivos parâmetros.

O contador **PendingAnswers** contabiliza o número de chamadas de abstrações de outros agentes, com parâmetros do tipo **out** ou **inout**, que foram disparadas pelo agente na última execução de sua regra de transição. Um agente somente pode re-executar sua regra de transição quando este contador estiver zerado.

O protocolo de comunicação entre os agentes impõe que se durante a execução de sua regra de transição, um agente a faz uma chamada a uma abstração de um outro agente b, os dados relativos a este ato são capturados pelo superagente do sistema, e o agente a prossegue imediatamente a execução de sua regra de transição. O efeito desta chamada somente se fará sentir na próxima transição.

O superagente de posse das informações recebidas do agente a, insere o pedido de execução da abstração na fila **RequestsReceived** do agente b. Se houver no pedido de execução parâmetros do tipo **out** ou **inout**, o contador **PendingAnswers** do agente a é incrementado de uma unidade pelo superagente. O contador será decrementado quando a resposta ao pedido do agente a for instalado em sua fila de **ReceivedAnswers**.

Antes de reiterar a execução de sua regra de transição, o agente retira uma a uma as atualizações contidas na sua fila **AnswersReceived** e realiza, na ordem da fila, as atualizações indicadas. Em seguida retira e executa, uma a uma, em ordem, todas as **action** relacionadas na fila **RequestsReceived**.

Estas operações de processamento das filas **AnswersReceived** e **RequestsReceived** são repetidas periodicamente pelo agente até que seu contador **PendingAnswers** torne-se zero, ocasião em que a regra de transição de estado do agente será novamente executada e todo o processo descrito acima se repete.

2.6 Expressão de Tipo

As expressões de tipos de Machĭna são as seguintes:

- tipos básicos: **Bool**, **Char**, **Int**, **Real**, **String** e intervalos;
- tipos compostos: listas, agentes, tuplas, conjuntos, uniões disjuntas, enumerações, funcionais e tipos definidos pelo programador;

- tipos genéricos: listas genéricas (**List**), conjuntos genéricos(**set**), agentes genéricos (**Agent**), tipo `?` e a união disjunta de todos os tipos;
- tipos arquivos: arquivos *stream* (**Input** e **Output**) e arquivos binários (**file** OF T).

A sintaxe de uma expressão de tipo é:

type-expression	::=	builtin-type | enumeration-type | compound-type | user-defined-type
	|	formal-type | abstraction-type | (type-expression) | type-variable
builtin-type	::=	basic-type | generic-type
basic-type	::=	**Bool** | **Char** | **Int** | **Real** | **String**
generic-type	::=	**List** | **Set** | **Agent** | **Input** | **Output**
compound-type	::=	union-type | tuple-type | list-type | file-type | agent-type
	|	functional-type | set-type | tree-type
type-identification	::=	type-designator | builtin-type
type-variable	::=	type-name

Nas regras e expressões de inspeção de tipos, por exemplo, na regra *with*, a identificação do tipo deve ser nominal, de acordo com a seguinte sintaxe de `type-identification` acima e nos demais contexto, vale a regra de equivalência estrutural de tipos.

2.7 Tipos Básicos

2.7.1 Tipo Bool

O tipo **Bool** possui os literais lógicos **true** e **false** e as seguintes operações:

- operações de atribuição, comparação. **false** é considerado **<** **true**;
- a operação unária de negação lógica, utilizando o operador **not**;
- as operações binárias que utilizam os operadores lógicos **and** (conjunção), **or** (disjunção) e **xor** (**or** exclusivo).

2.7.2 Tipo Char

O tipo **Char** possui literais caracteres e as seguintes operações:

- operações de atribuição e comparação;
- `ord(c:`**Char**`):`**Int**, que retorna o código ASCII do caractere `c`;
- `chr(i:`**Int**`):`**Char**, que retorna o caractere com o código ASCII `i`;
- `succ(c:`**Char**`)`, que retorna o sucessor de `c` ou **undef**;
- `pred(c:`**Char**`)`, que retorna o antecessor de `c` ou **undef**.

2.7.3 Tipo Int

O tipo **Int** compreende os literais inteiros e as operações:

- operações de atribuição e comparação;
- operação unária de negação aritmética, utilizando o operador `-`;
- as operações binárias de adição (`+`), subtração (`-`), multiplicação (`*`), divisão (`/`) e resto da divisão (`%`);
- `abs(Int):Int` – valor absoluto de um número inteiro;
- `max(Int,Int):Int` – máximo entre dois inteiros;
- `min(Int,Int):Int` – mínimo entre dois inteiros;
- `sqr(Int):Int` – quadrado de um número inteiro.

2.7.4 Tipo Real

O tipo **Real** compreende os literais reais e as operações:

- operações de atribuição e comparação;
- operação unária de negação, utilizando o operador `-`;
- operações binárias de adição (`+`), subtração (`-`), multiplicação (`*`) e divisão (`/`);
- `abs(Real):Real`, o valor absoluto de um número real;
- `max(Real,Real):Real`, o máximo entre dois números reais;
- `min(Real,Real):Real`, o mínimo entre dois números reais;
- `sqr(Real):Real`, o quadrado de um número real;
- `sqrt(Real):Real`, a raiz quadrada de um número real;
- `integer(Real):Int`, conversão de número real para inteiro, utilizando truncamento;
- `real(Int):Real`, conversão de número inteiro para real.

2.7.5 Tipo String

O tipo **String** possui os literais cadeias de caracteres e as seguintes operações:

- operação binária de concatenação, utilizado o operador `+`;

- as operações do tipo **Int-> Char**, para acesso ou modificação de caracteres de uma cadeia, e.g., `s(k) := s(j)` copia o caractere da posição `j` de `s` para a posição `k`, para inteiros `k` e `j`, 1 <= `j`, `k` <= `length`(`s`), sendo `s`:**String**;
- `length(s:` **String**`):`**Int**, que retorna o comprimento da cadeia de caracteres `s`;
- `equals(s1:`**String**`, s2:`**String**`):`**Bool**, que informa se as cadeias `s1` e `s2` são iguais;
- `compareTo(s1:`**String**`, s2:`**String**`):`**Int**, que retorna `-1` se `s1` for menor que `s2`, `0` se iguais, e `1` se maior.

2.8 Tipo Enumeração

Uma expressão de tipos enumeração é da forma **enum** $\{a_1, \cdots, a_n\}$, onde a_i são os identificadores que representam as constantes simbólicas da enumeração, como no exemplo:

```
RGBColor = enum { RED, GREEN, BLUE }.
```

Sintaxe:

enumeration-type	::=	[**public**] **enum** "{"constant-name { , constant-name } "}"
constant-expression	::=	expression

Enumerações diferentes devem definir identificadores diferentes. Isto significa que não é permitida uma seqüência de declarações da forma:

```
X = enum { A, B, C }
Y = enum { A, E, I, O, U }
```

pois o identificador `A` está sendo definido em duas enumerações diferentes.

As operações de um tipo enumeração são:

- atribuição e comparação (`=, !=, <. <=. >, >=`);
- `succ(c:T)`, que retorna o sucessor de `c` na enumeração `T` ou **undef**;
- `pred(c:T)`, que retorna o antecessor de `c` na enumeração `T` ou **undef**.

O tipo pré-definido **State** é equivalente a **State = enum** {**unborn, new, running, blocked, dead**} e representa os possíveis estados de um agente:

- **unborn**- agente declarado, mas ainda não criado;
- **new**- agente criado, mas ainda não disparado;

- **running**- agente em execução e não bloqueado;
- **blocked**- agente em execução, mas bloqueado à espera de alguma mensagem;
- **dead**- agente destruído.

A expressão `a.state` do tipo **state**, informa o estado corrente da agente `a`.

2.9 Tipo União Disjunta

Uma expressão de tipos união disjunta é da forma $T_1|\cdots|T_n$, onde T_i são expressões de tipos. Por exemplo, o tipo `Number =` **Int** `|` **Real** é a união disjunta de **Int** e **Real**. As parcelas da união são chamadas de projeções do tipo união.

Sintaxe:
union-type ::= ? | (type-expression "|" type-expression)

A projeção do valor de uma união disjunta, isto é, a obtenção do valor armazenado em uma variável cujo tipo é união disjunta, é feita utilizando a regra *with*. A injeção, isto é, a criação de uma união disjunta a partir de um valor cujo tipo é um dos constituintes da união, é feita pela atribuição direta, conforme definido na disciplina de tipos de Machĭna.

O tipo `?` é definido como a união disjunta de todos os tipos possíveis, com o valor *default* igual a `undef`. Isto significa que, se `x` é do tipo `?`, então `x` pode receber valores de quaisquer tipos. O tipo `?` só possui as operações de atribuição e comparação por `=` ou `!=`. Conversão para o tipo do valor armazenado deve ser feita via regra **with** e expressão **is**.

2.10 Tipo Tupla

Uma expressão de tipos tupla é da forma **tuple**$(f_1 : T_1, \cdots, f_n : T_n)$, onde f_i são os nomes dos campos da tupla e T_i são os seus respectivos tipos.

Sintaxe:
tuple-type ::= **tuple**([tuple-elements])
tuple-elements ::= tuple-element { , tuple-element }
tuple-element ::= [function-name :] type-expression

As tuplas podem ter seus componentes rotulados ou não. Pode-se utilizar a tupla vazia, `( )`, em definições recursivas de tipos. Por exemplo, considere a declaração:

```
Tree = tuple( ) | tuple(info:Info, left:Tree, right:Tree).
```

Uma expressão da forma `T(t1, t2, t3)` cria uma tupla do tipo `T`, formada pelos elementos `t1`, `t2` e `t3`.

As operações sobre tupla são: atribuição e seleção de seus componentes via qualificacão, e.g., `q.x` no exemplo abaixo:

```
...
p, q:tuple(x:Int, y:Int);
...
q := p;
q.x := p.y
...
```

2.11 Tipo Conjunto

O construtor de tipos "**set of**" é utilizado para denotar o tipo conjunto de elementos de um mesmo tipo. Por exemplo

```
Intset = set of Int
```

é um tipo cujos elementos são conjuntos de números inteiros. O tipo pré-definido **set** é definido como

```
Set = set of ?.
```

Isto significa que **set** é um conjunto genérico, onde seus elementos não precisam de ser do mesmo tipo.

Sintaxe:
set-type ::= **set of** type-expression

Assim como listas, conjuntos podem ser genéricos ou de um tipo específico. As operações envolvendo conjuntos são:

`s1 + s2`	união
`s1 - s2`	diferença
`s1 * s2`	interseção
`x in s`	pertinência
`s(x)`	pertinência
`s1 relop s2`	inclusão (`<=`), igualdade (`=`), etc.

Um agregado conjunto é da forma

$$\{t_1, t_2, ..., t_n\}.$$

Se todas as expressões forem do mesmo tipo X, então o tipo da expressão toda é **set of** X; caso contrário, o tipo será `Set`.

2.12 Tipo Lista

A linguagem Machĭna possui dois tipos de listas: listas genéricas e listas de elementos de um tipo específico. Uma lista vazia é dada pelo literal **nil**. O construtor de tipo "**list of**" é utilizado para denotar o tipo lista de elementos de um mesmo tipo. Por exemplo, `Intlist = list of Int` é um tipo cujos elementos são listas de inteiros.

Sintaxe:
list-type ::= **list of** type-expression

O tipo pré-definido **List** é equivalente a **list = list of** `?`. Desta forma, **List** representa uma lista genérica, onde seus elementos não precisam ser do mesmo tipo.

Sobre listas, estão definidas as seguintes operações:

`x :: listA`	retorna uma lista cuja cabeça é `x` e cuja cauda é `listA`
`listA + listB`	retorna a concatenação das listas `listA` e `listB`
`listA :: x`	retorna a lista dada acrescida de `x` no fim
`head(listA)`	retorna o elemento na cabeça de `listA`
`tail(listA)`	retorna a cauda de `listA`
`length(listA)`	retorna o comprimento de `listA`
`x in listA`	informa se o elemento `x` está na lista `listA`

onde `x` tem tipo `T` e `listA e ListB` são do tipo **list of** `T`.

Em tempo de compilação, são feitos testes para consistência e inferência de tipos. As regras para consistência de tipos são as seguintes:

1. nas operações `x::listA` e `listA::x`, se `x` possuir o tipo `X`, então `listA` deve ser do tipo **List** ou **list of** Y, tal que `X` seja compatível com `Y`;

2. na operação `listA + listB`, `listA` e `listB` devem ser listas de elementos do mesmo tipo, a menos que uma delas seja do tipo **List**; neste caso o resultado é do tipo **List**;

3. se `listA` for do tipo **list of** X, então `head(listA)` é do tipo `X`; se `listA` for do tipo **List**, então `head(listA)` será do tipo "`?`";

4. a expressão `tail(listA)` tem o mesmo tipo de `listA`.

Um agregado do tipo lista tem a forma $[t_1, t_2, ..., t_n]$. Se todas as expressões forem do mesmo tipo X, então o tipo da expressão toda é **list of** X; senão, o tipo será **List**.

O tipo genérico **List** é definido como "**list of** `?`".

2.13 Tipo Nodo de Árvore

O tipo nodo de árvore denota a estrutura de nodos de uma árvore de sintaxe abstrata definida pelo programador. A partir da estrutura de nodo, pode-se decompor os elementos de uma árvore de derivação por meio de casamento de padrão e associação de nome de funções a cada um dos elementos de estrutura. O casamento de padrão pode ser feito com a regra ou expressão *with*.

Sintaxe:
tree-type ::= "[" { tree-element } "]"
tree-element ::= type-identification | string-literal

Abaixo, são apresentados o tipo `Cmd` que define a estrutura dos nodos de árvore de comandos para uma linguagem de programação:

```
type Cmd = ["while" Exp "do" Cmd "end"]
         | [Id ":=" Exp]
         | ["if" Exp "then" Cmd "else" Cmd "end"] ;
type Exp  = [Exp "+" Exp]
         | [Id] ;
type Id   = Token;
c, c1 : Cmd;
exp   : Exp;
id    : Id;
...
with c as [id ":=" exp] => ... id ... exp
       as ["while" exp "do" c1 "end"] => ... exp ... c1 ...
end
```

2.14 Tipo Funcional

O tipo funcional é um tipo cujos valores são funções entre domínios dados. Por exemplo, o tipo `Transition` definido por `Transition = (MachineState,Element) -> State` é o tipo das funções de `MachineState` × `Element` em `MachineState`. Não é permitido em Machĭna a definição de funções de ordem mais alta.

Sintaxe:
functional-type ::= type-expression `->` type-expression
formal-type ::= type-name

As operações do tipo funcional é a obtenção do valor da função em um ponto e atualização do valor associado a um ponto.

2.15 Tipo Agente

O construtor de tipos **agent of** `M` é utilizado para denotar o tipo dos agentes que executam a regra do módulo `M`. Existe também o tipo pré-definido **Agent**, que representa agentes de quaisquer tipos. As operações que podem ser feitas sobre agentes são a criação, o disparo da execução, a eliminação, a atribuição, a passagem como parâmetro, o retorno de função e a comparação por igualdade.

Sintaxe:
agent-type ::= **agent of** module-name

O construtor de tipos **agent of** recebe um nome de módulo `M` e define o tipo dos agentes que executam a regra de transição de `M`.

O tipo agente genérico **Agent** compreende agentes de qualquer um dos módulos conhecidos na especificação.

2.16 Tipo Abstração de Regras

Ações podem ser passadas como parâmetros para outras ações e atribuídas a variáveis.

Sintaxe:
abstraction-type ::= **action** [(parameter-type { , parameter-type })]
parameter-type ::= [passing-type] [parameter-name :] type-expression
passing-type ::= **in** | **out** | **inout**

2.17 Tipo Arquivo

Arquivos são declarados via o construtor de tipo **file of**, que recebe como argumento o tipo dos registros do arquivo.

Sintaxe:
file-type ::= **file of** type-expression

A qualquer arquivo, pode-se aplicar a função `eof(f:file of ?):Bool`, que recebe como argumento um arquivo de qualquer tipo e retorna **true**, se o arquivo estiver posicionado no fim, ou **false**, no caso contrário. Se um dado arquivo `f` estiver fechado, `eof(f)` returna **true**.

Todas as ações de entrada/saída possuem, além de outros, um parâmetro **out** `status:Int`. Esta parâmetro serve para informar o código da condição de erro eventualmente ocorrida durante a execução da operação. O valor de `status` devolvido pela operação pode ser:

0	operação realizada com sucesso
1	tentativa de ler ou escrever em arquivo fechado
2	tentativa de abrir arquivo já aberto
3	tentativa de fechar arquivo já fechado
4	erro de leitura no arquivo
5	erro de escrita no arquivo
6	tentativa de ler além do fim do arquivo
7	erro de conversão de tipo na operação

Para manipulação de arquivos do tipo fluxo (*stream*), existem os tipos pré-definidos **Input**, **Output**. O tipo pré-definido **Input** representa um arquivo texto para leitura com caracteres de leiaute. As operações que podem ser feitas sobre este tipo são:

- `open(out infile:Input, in s:String, out status:Int)` – ação que abre, para leitura, o arquivo de nome `s`, retorna uma referência para o arquivo em `infile`;
- `close(in infile:Input,out status:Int)` – ação que fecha o arquivo referenciado por `infile`;
- `readInt(in infile:Input, out i:Int, out status:Int)` – ação que lê um valor inteiro do arquivo `infile` e retorna em i;
- `readChar(in infile:Input, out c:Char, out status:Int)` – ação que lê um caractere do arquivo `infile` e retorna em c;
- `readReal(in infile:Input, out r:Real, out status:Int)` – ação que lê um número ponto-flutuante do arquivo `infile` e retorna em r;
- `readString(in infile:Input, out s:String, out status:Int)` – ação que lê uma cadeia de caracteres do arquivo `infile` e retorna s.

O tipo pré-definido **Output** é análogo ao tipo **Input** e representa um arquivo texto, para escrita, com caracteres de leiaute. As operações que podem ser feitas sobre este tipo são:

- `open(out outfile:Output, in s:String, out status:Int)` – ação que abre, para escrita, o arquivo de nome `s`, retornando uma referência para o arquivo aberto em outfile;
- `close(in outfile:Output, out status:Int)` – ação que fecha o arquivo referenciado por `outfile`;
- `writeInt(in outfile:Output, in i:Int, out status:Int)` – ação que escreve o valor de `i` no arquivo `outfile`;
- `writeChar(in outfile:Output, in c:Char, out status:Int)` – ação que escreve o valor de `c` no arquivo `outfile`;
- `writeReal(in outfile:Output, in r:Real, out status:Int)` – ação que escreve o valor de `r` no arquivo `outfile`;

- `writeString(in outfile:Output, in s:String, out status:Int)` – ação que escreve o valor de `s` no arquivo `outfile`.

Uma expressão de tipo **file of** T, onde T é um tipo, cria um tipo que representa arquivos, cujos elementos são do tipo T. Estes arquivos são *binários*, com acesso *seqüencial* ou *direto*. As operações aplicáveis a estes arquivos são:

- `open(out f:file of T, s:String, readOnly:Bool, out status:Int)` – ação pré-definida que abre, para leitura ou para escrita, o arquivo de nome `s`, colocando uma referência para este arquivo no parâmetro `f`; o arquivo será de leitura de o parâmetro `readOnly` for verdadeiro, caso contrário, será de escrita;
- `close(in f:file of T, out status:Int)` – ação pré-definida, que fecha o arquivo referenciado por `f`;
- `read(in f:file of T, out t:T, out status:Int)` – lê um valor do tipo `T`, do arquivo referenciado por `f` e retorna em t;
- `write(in f:file of T, in t:T, out status:Int)` – escreve `t`, no arquivo referenciado por `f`;

É importante salientar que, devido ao modelo algébrico subjacente à Machĭna, qualquer operação de leitura em arquivos só é percebida no próximo estado. Da mesma forma, qualquer chamada a `status` em um mesmo estado retorna o mesmo valor, que é o *status* do arquivo no passo anterior. Assim como qualquer ação de Machĭna, a ordem de execução de duas operações sobre arquivos no mesmo estado não é garantida. Por exemplo, considere o código abaixo:

```
    f : Input ;
    g : Output;
    s : Int;
...
begin:
    readInt(f,a,s); readInt(f,a,s); readInt(f,b,s);
    writeInt(g,x,s); writeInt(g,y,s);
end
```

Serão lidos três valores do arquivo `f` e escrito dois valores no arquivo `g`. Entretanto, não há como determinar qual será o valor de `a` e qual será o valor de `b`, pois isto dependerá da ordem em que o compilador escolher executar este código. Da mesma forma, não é possível determinar qual valor será escrito antes em `g`, se o valor de `x` ou se o valor de `y`.

2.18 Novos Tipos

Tipos precisam ser declarados antes de serem usados, exceto na escrita de tipos mutuamente recursivos, quando Machĭna permite que, na declaração de um tipo, apareçam nomes de

tipos que ainda não foram declarados. Tipos mutuamente recursivos, contudo, devem ser declarados dentro de um mesmo módulo, porque não se permite circularidade de dependência, via cláusula **include**, entre módulos. Dependência mútua é permitida somente via **import**.

Em uma definição de tipos, pode haver também uma cláusula **default**, que permite estabelecer o valor *default* para inicialização automática de funções do tipo dado.

Uma seção de declaração de tipos tem a seguinte sintaxe:

type-declaration	::=	**type** type-declarator [= type-denotation]
type-declarator	::=	type-name [(type-parameters)]
type-parameters	::=	type-parameter { , type-parameter }
type-parameter	::=	parameter-name
type-denotation	::=	type-expression [**default** expression]

Uma definição de tipos em Machĭna pode ser parametrizada da forma:

type `T(`$t_1, t_2, ..., t_n$`) =` E **default** e**;**

onde T é um nome de tipo, $(t_1, t_2, ..., t_n)$ é uma lista opcional de parâmetros que denotam tipos, E é uma expressão de tipos, e e é uma expressão, cujo tipo é compatível com E.

Tipos parametrizados devem ser instanciados no momento do uso, de acordo com a sintaxe:

user-defined-type	::=	type-designator [(type-arguments)]
type-arguments	::=	type-argument { , type-argument }
type-argument	::=	type-expression

Por exemplo, os tipos representados pelo parâmetros formais de tipo $(t_1, t_2, ..., t_n)$ devem ser fornecidos nas declarações de funções do tipo T, como nas declarações de `x` e `y` em:

algebra:
  `x:` `T(`**Int,Real,...,String**`);`
  `y:` `T(`**Real,Real,...,Int**`);`
  ...
abstractions:
  public action `p(a:T(`$t_1, t_2, ..., t_n$`) , b:`t_1`)` **is** ...

A instanciação de um tipo parametrizado causa a associação dos tipos parâmetros formais aos argumentos do tipo declarado.

Nas assinaturas ou cabeçalhos de **action**s e de funções, argumentos de tipos parametrizados podem ser variáveis de tipos, como t_1 na definição da ação `p` acima. Neste caso, todos as ocorrências de uma mesma variável de tipo em uma mesma assinatura devem ser associadas a um mesmo tipo. Por exemplo, no caso da chamada `p(v,w)`, o tipo declarado para `w` deve ser o mesmo tipo de primeiro elemento da tupla `v`.

2.19 Valor *Default* de Cada Tipo

O valor *default* de um tipo pode ser definido por meio da cláusula **default** na declaração do tipo. Para os tipos primitivos e para os tipos não recursivos construídos por meio dos construtores de tipo da linguagem, o valor *default* é dado da seguinte forma:

- para o tipo **Bool**, o valor *default* é **false**;
- para o tipo **Char**, o valor *default* é $'\backslash 0'$, i.e., o caractere cujo código ASCII é igual a `0`;
- para o tipo **Int**, o valor *default* é o número `0`;
- para o tipo **Real**, o valor *default* é o número `0.0`;
- para o tipo **String**, o valor *default* é a cadeia de caracteres vazia;
- para o tipo `?`, o valor default é o elemento **undef**;
- para todo tipo T, o valor default do tipo **list of** T é igual a `[]`, isto é, a lista vazia; este também é o valor default do tipo **List**;
- para o tipo tupla $T = (T_1, T_2, \cdots, T_n)$, o valor default é dado por

  $$T(\texttt{default}(T_1), \texttt{default}(T_2), \cdots, \texttt{default}(T_n))$$

  onde T é o nome dado na declaração;
- para o tipo união disjunta $T = T_1|T_2|\cdots|T_n$, o valor default é igual ao valor default de T_1, isto é, $\texttt{default}(T_1)$;
- não há valores default para agentes, arquivos ou funções de tipos recursivos.

Considere as declarações:

```
type D = Int default -1;
x : Int;
y : D;
type T = tuple (a,b:Int; x:A);
type A = Real | String;
type C = list of B;
type B =  tuple (a:Int; c:C);
```

O valor inicial de `x` é igual a 0, pois este é o valor default do tipo **Int**. O valor inicial de `y` é igual a -1, pois este é o valor default do tipo `D`. O valor default do tipo `A` é igual a `0.0`, que é o valor default de **Real**, que é o primeiro elemento da união. O valor default do tipo `T` é igual a `T(0,0,0.0)`, que é o agregado tupla construído a partir dos valores default do tipo. Os tipos `B` e `C` não possuem valor default, pois são mutuamente recursivos.

2.20 Visibilidade de Componentes de Tipo

Quando um tipo declarado como público em um módulo é incluído em outro módulo, somente seu nome se torna visível no ponto de inclusão, mas não a sua estrutura. Exceção para enumeração, quando for necessário revelar seus elementos. Para isso, Machĭna define a regra que, para exportar os *elementos de uma enumeração*, escreve-se a palavra-chave **public** antes da palavra **enum**. Por exemplo, no código a seguir, são visíveis externamente a `M` os nomes de `T`, `Q`, `X`, `Y`, `Z`, ao passo que os nomes `A`, `B` e `C` não são visíveis, pois pertencem a uma enumeração não-pública.

```
module M
algebra:
   public type  T = enum {A,B,C};
   public type  Q = public enum {X,Y,Z};
end
```

2.21 Disciplina de Tipos

A disciplina de tipos de Machĭna utiliza o critério de equivalência e compatibilidade estruturais de tipo em todas as construções, exceto em três casos específicos, a regra *with*, expressão *with* e expressão *is*. Nestes casos, utiliza equivalência nominal, porque este é o propósito destas construções.

Em Machĭna, tudo que diz respeito a tipos é decidido em tempo de compilação, de modo que, a menos das uniões disjuntas, os objetos não levam para a execução informações sobre os seus tipos.

Diz-se que um tipo A é compatível com um tipo B, se uma expressão do tipo A puder ser usada em um lugar onde é esperada uma expressão do tipo B. Por exemplo, se definirmos que x é do tipo A e y é do tipo B, a atribuição $x := y$ só é possível se B for compatível com A.

O tipo A é *compatível* com o tipo B se e somente se uma das seguintes condições se aplicar:

- A e B são nomes de tipos idênticos;
- B é o tipo `?`;
- B é uma expressão de tipos e A é um nome de tipos, cuja definição visível é $A = d$ ou $A = A_1$, $A_1 = A_2$, $\cdots$, $A_{n-1} = A_n$, $A_n = d$, onde A_i, para $1 \leq i \leq n$, é um nome de tipo e a expressão de tipos d é compatível com a expressão de tipos B;

- A é uma expressão de tipos e B é um nome de tipos, cuja definição visível é $B = d$ ou $B = B_1$, $B_1 = B_2$, $\cdots$ $B_{n-1} = B_n$, $B_n = d$, onde B_i, para $1 \leq i \leq n$, é um nome de tipo, e a expressão de tipos A é compatível com a expressão de tipos d;
- A e B são tipos de listas, A é da forma **list of** A_1 e B é da forma **list of** B_1, e A_1 é compatível com B_1, ou B é o tipo **List**, ou `A` é o tipo polimórfico de listas vazias;
- A e B são tipos de tuplas, A é da forma $(A_1, ..., A_n)$, B é da forma $(B_1, ..., B_n)$ e, para $1 \leq i \leq n$, A_i é compatível com B_i, e B_i é um membro privado se A_i o for. Nomes de campos não são relevantes para a determinação da compatibilidade;
- B é uma união disjunta da forma $B_1| \cdots |B_n$ e A é compatível com algum B_i, $1 \leq i \leq n$;
- A é uma união disjunta da forma $A_1| \cdots |A_m$, B é uma união disjunta da forma $B_1| \cdots |B_n$ e todo A_i, $1 \leq i \leq m$, é compatível com algum B_j, $1 \leq j \leq n$;
- A e B são tipos de agentes do mesmo módulo, ou B é o tipo **Agent**;
- A e B são tipos funcionais, A é da forma A_1`->`A_2, B é da forma B_1`->`B_2, B_1 é compatível com A_1 e A_2 é compatível com B_2.
- A e B são tipos abstração de **action**, A é da forma **action**$(A_1, \cdots, A_n)$, B é da forma **action**$(B_1, \cdots, B_n)$, para $1 \leq i \leq n$, A_i é compatível com B_i

2.22 Declaração de Funções

A sintaxe de definição de funções e relações em Machĭna é a seguinte:

function-declaration	::=	[class-modifier] declared-element : type-expression [:= expression]
class-modifier	::=	**static** | **derived** | **dynamic**
declared-element	::=	function-name { , function-name }
	|	function-name (function-parameters)
function-parameters	::=	function-parameter { , function-parameter }
function-parameter	::=	parameter-name : type-expression

Os parâmetros de uma função podem ser de qualquer tipo, exceto funcionais e abstrações de regras. Isto é, não se permite funções ordem mais alta e nem passagem de acões como parâmetro.

As funções, quanto sua natureza, podem ser:

- estáticas – são as que não podem sofrer atualizações e nem acessar funções dinâmicas, derivadas ou externas. São identificadas pelo modificador **static**;
- derivadas – são funções que não podem sofrer atualizações, mas podem acessar funções dinâmicas, derivadas ou externas. São identificadas pelo modificador **derived**.
- externas – são funções definidas e atualizadas no ambiente externo. São identificadas pelo modificador **external**.

- dinâmicas – são as funções identificadas pelo modificador **dynamic** e aquelas sem modificadores de classe, que são assumidas dinâmicas por default.

Na definição de uma função estática ou dinâmica, define-se sua assinatura e, se desejado, também o seu valor inicial. No exemplo abaixo, a função `f` é uma função dinâmica e `g` uma função estática, ambas com assinatura dada por **Char** × **Char** → **Int** e valor inicial $\lambda(x, y).1$.

```
module M
  ...
  algebra:
    external square:Int-> Int;
    f(a:Char, b:Char):Int:= 1;
    static g(x:Char, y:Char):Int:= 1;
    static h(x:Char, y:Char):Int;
    derived mysquare(x:Int):Int:= square(x);
  ...
end
```

Quando uma função estática ou dinâmica é declarada sem um valor de inicialização explícito, seu valor inicial será o valor *default* do seu tipo de saída (contradomínio). No exemplo acima, como a função `h` não recebe valor inicial, ela será definida por $\texttt{h} = \lambda xy.0$, pois 0 é o valor *default* de **Int**. Por outro lado, definições de funções derivadas possuem obrigatoriamente o corpo para ser avaliado. No exemplo acima, definimos a função `mysquare`, que retorna o quadrado de um número inteiro, usando a função externa `square`.

2.23 Inicialização de Funções Dinâmicas e Estáticas

Valores iniciais de funções estáticas e dinâmicas podem também ser definidos (ou redefinidos) na seção de inicialização, utilizando a sintaxe de regras de transição de Machĭna, como mostrado acima na definiçao da função `f`. A regra de transição da seção **initial state:**é executada uma única vez por agente, antes de se iniciar a execução repetida da regra de transição do agente. As regras de inicialização e de transição de módulos secundários somente são executadas por agentes criados a partir destes módulos.

```
module M
  ...
  algebra:
    f(a:Char, b:Char):Int:= 1;
    static g(x:Char, y:Char):Int:= 0 ;
    static h(x:Char, y:Char):Int;
  initial state:
      forall x:Chardo f(x) := f(x) + 10 ; h(x,y) := 2 end;
      forall x:Char,y:Char do g(x,y) := g(x,y) + 1; h(x,y) := 2 end;
end
```

2.24 Funções Externas

A sintaxe de definição de funções externas em Machĭna é a seguinte:

external-declaration	::=	**external** external-function : type-expression
external-function	::=	function-name { , function-name }
	|	function-name (external-parameters)
external-parameters	::=	external-parameter { , external-parameter }
external-parameter	::=	parameter-name : type-expression

Uma função externa é uma função definida externamente a qualquer módulo Machĭna. Como o seu valor é definido pelo ambiente externo, elas não possuem um corpo para ser avaliado. Desta forma, a definição de uma função externa contém somente a especificação de sua assinatura, como `fext` em **external** `fext(x :Int):Int;`.

Funções externas podem ser escritas em C conforme o protocolo definido para a passagem de parâmetros e o valor de retorno das funções, de modo que possamos obter precisão na avaliação das funções externas. O protocolo é o seguinte:

- para cada parâmetro da função externa, existe um parâmetro na função escrita em C;
- a ordem dos parâmetros nas duas funções deve ser a mesma;
- os tipos dos parâmetros e de retorno na função escrita em C devem seguir o tipo do cabeçalho definido para a função externa, conforme a seguinte equivalência:
  - **Bool**, **Char** e enumerações equivalem ao tipo `char` de C;
  - **Int** equivale ao tipo `int` de C;
  - **Real** equivale ao tipo `double` de C;
  - **String** equivale ao tipo `char*` de C;
  - **List** é um tipo que equivale a uma de lista de elementos dos tipos;
  - um tipo tupla equivale a uma estrutura de C, de modo que os campos da tupla seguem este protocolo de equivalência com os campos da estrutura na implementação em C.

Por exemplo, os cabeçalhos de funções externas abaixo são equivalentes aos cabeçalhos de funções em C dados em seguida.

```
/* Em Machĭna */
algebra:
  type T = (a:Int; b:Bool);
  a,b :Int;
  c:Char;
  d:T;
  external f(x:Int):Int;
  external g(x,y:Int;c:Char):Bool;
  external h(a:Int;b:T):T;
```

```
transition:
...  x := f(10) ...
...  y := g(a,b,c) ...
...  z := h(1,d) ...
/* Protótipos em C */
typedef struct{int t1; char t2;} *T;
int f(int x);
char g(int x, int y, char c);
T h(int a, T b);
```

2.25 Declaração de Abstrações de Regras

Abstrações de regras são um recurso apropriado para se definir operações de tipos abstratos de dados. A sintaxe das abstrações de regra é:

rule-abstraction	::=	action-declaration
action-declaration	::=	**action** rule-declaration
rule-declaration	::=	rule-heading **is** contract rule-body **end** [rule-name]
rule-heading	::=	rule-name [(rule-parameters)]
rule-parameters	::=	rule-parameter { , rule-parameter }
rule-parameter	::=	[passing-mode] parameter-name : type-expression
contract	::=	[pre-condition] [pos-condition]
pre-condition	::=	**require** expression
pos-condition	::=	**ensure** expression
rule-body	::=	one-pass-transition | repeating-transition
one-pass-transition	::=	local-declarations [init-state-part]**begin:** single-transition
	|	[**begin:**] single-transition
repeating-transition	::=	[local-declarations[init-state-part]] **loop:** main-transition
local-declarations	::=	**algebra:** { local-algebra-section }
local-algebra-section	::=	local-type | local-function | local-external
local-type	::=	type-declaration ;
local-function	::=	function-declaration ;
local-external	::=	external-declaration ;
passing-mode	::=	**in** | **out** | **inout**

Uma ação poder receber qualquer tipo de parâmetro, inclusive outras ações.

O corpo de uma ação pode ser uma execução repetitiva, se marcado por **loop:** ou de execução unitária, quando marcado com **begin:**. Na ausência de das marca **loop:** ou **begin:**, procede-se com execução unitária.

Durante a execução, as eventuais atualizações têm efeito apenas local, inclusive para os parâmetros e funções dinâmicas globais que ocorrerem no corpo da abstração. As atual-

izações ocorridas na abstracão somente são percebidas pela regra de transição do agente no passo seguinte, da mesma forma que ocorre com uma regra de atualização de módulo.

Para alcançar este efeito, a primeira providência de uma uma ação repetitiva, marcada com **loop:**, é fazer uma cópia local de todos os parâmetros recebidos e de todas as funções dinâmicas globais atualizadas no seu corpo. Funções dinâmicas com acesso somente de leitura no corpo da abstração não são copiadas. A regra de transição das iterações operam diretamente com estas cópias locais e com as funções globais só de leitura, e no fim da ação definida pela abstração, a lista de atualização obtida para as cópias locais são refletidas na lista de atualização corrente do ponto de chamada da abstração. Do ponto de vista do módulo que faz a chamada, tudo se passa como se as alterações geradas pela ação houvessem ocorrido em um único passo.

No caso de ação sem a marca **loop:**, a cópia local citada acima não se faz necessária. Neste caso, executa-se uma única vez a regra de transição no corpo da abstração e retorna-se ao ponto de chamada com a lista de atualizações produzidas para parâmetros e funções globais à ação. Neste caso, diferentemente das iterações, não se fazem cópias das funções globais atualizadas na abstração.

O esqueleto abaixo ilustra três tipos de abstrações de ação:

```
...
z : Tm; ...
action a(a1 : T1, ··· , ak : Tk) is D ;R end;
action b(b1 : T1, ··· , bk : Tk) is begin: D ;R end;
action c(c1 : T1, ··· , ck : Tk) is loop: D ;R end;
action r(x : Tm, y : action(T1, ··· , Tk)) is loop: ... y(...); ... end;
...
r(z,a);
...
```

onde, para $1 \leq i \leq k$, a_i, b_i e c_i são identificadores, T_i, nomes de tipo, D representam declarações de escopo local e R é uma regra de transição. Se a regra R contiver alguma operação de atualização de um dos parâmetros p_i, então dizemos que p_i é um parâmetro de saída da abstração. Note que ação r aceita uma ação do tipo indicado como parâmetro.

As regras para inferência dos tipos dos parâmetros de uma abstração de regra é idêntica à dedução dos tipos dos parâmetros de uma função. Qualquer inconsistência de tipos causará um erro de compilação. Os modos de passagem de parâmetros são **in**, **out** e **inout**. Se omitido a especificação do modo de passagem, assume-se **inout**. A passagem de parâmetros para abstrações **out** ou **inout** é por nome (*call by name*). Desta maneira, as atualizações destes parâmetros no corpo da abstração são refletidos nos argumentos de chamada. Os parâmetros do tipo **in** são passados por valor.

Os elementos declarados locais a uma abstração sobrevivem de uma chamada à seguinte, preservando, portanto, seus valores finais entre uma chamada e a seguinte. Na primeira

execução da ação, os valores iniciais dos elementos locais são os valores default de seus tipos. As inicializações de funções contidas na declaração, se houver, são executadas sempre que a abstração for chamada, reiniciando seus valores; caso contrário, os valores assumidos no fim da última chamada estarão disponíveis.

Não são permitidas chamadas recursivas a abstração de regra, mas uma abstração pode chamar qualquer outra, desde que não gere circularidade.

2.26 Regras de Transição de Estado

A transição principal de um módulo ou do corpo de abstrações definidas em um módulo pode ser uma regra de transição de um único passo ou então de uma seqüência de passos de execução. No caso de transição de um único passo, a mesma regra é executada repetidamente pelo agente até que a condição de terminação seja atingida. Na transição de múltiplos passos, cada um dos passos indicados pelas cláusulas **step** é executado em sequência, começando-se com **step := 1**, sendo **step** uma variável pré-declarada com o tipo **Int**. O valor de **step** não pode ser alterado por atribuição direta. Para sua alteração dentro da regra de transição, há uma outra variável implícita inteira pré-definida, denominada **next**, cujo valor inicial é 1.

No início da execução de cada passo, essa variável **next** é automaticamente incrementada, e no fim de toda transição, faz-se automaticamente **step** := **next**. A próxima transição a ser executada é escolhida comparando-se o valor corrente da variável **step** com a constante inteira que rotula cada uma das regras de cada passo. Se não houver um rótulo especificado para um dado passo, nada é feito, exceto o incremento de **next** e atualização de **step**, e o passo seguinte é iniciado. O valor de **step** pode ser explicitamente alterado em um passo, atribuindo-se um valor à variável **next**, o qual então prevalece sobre o incremento automático, permitindo que se force a re-execução da regra a partir de determinado passo. A transição é dita concluída após a execução do último passo. Neste caso, o valor corrente de **step** supera o rótulo da última cláusula. Se a execução da regra for repetida, isto é, se a transição estiver marcada com **transition:** ou **loop:** as variáveis **next** e **step** são reiniciadas com valor 1 e a execução da regra reiniciada e o mesmo processo se repete.

Considere a seguinte regra de transição de passos múltiplos, e suponha que `R1, R2, R3, Ra` e `Rb` não contenham atribuição a **next**:

```
transition:
   step 1 : R1;
   step 2 : R2;
   step 4 : if ... then Ra; next := 2 else Rb end;
   step 6 : R3;
end
```

Neste exemplo, tem-se inicialmente a execução de `R1` seguida da de `R2`. No passo 3 nada é feito, exceto incrementar **next**. No passo 4, se a parte **then** for escolhida, executam-se as ações indicadas e volta-se ao passo 2; caso contrário, prossegue-se no passo seguinte, que é o

5, no qual nada é feito, exceto avançar para o passo seguinte. Após a finalização do passo 6, a execução da transição é re-iniciada com **next**=1 e **step**=1.

A transição abaixo, opera com o exemplo acima, exceto que após o passo **step** 6, a sua execução é concluída.

```
begin:
   step 1 : R1;
   step 2 : R2;
   step 4 : if ... then Ra; next := 2 else Rb end;
   step 6 : R3;
end
```

As regras de transição de Machĭna podem ser dos seguintes tipos: (i) Regras Básicas: Atualização, Blocos e Abreviaturas; (ii) Regras Condicionais; (iii) Regras de Universalização; (iv) Chamadas de Abstração; (v) Regras de Manipulação de Agentes.

A sintaxe da regra de uma transição é:

main-transition	::=	stepwise-transition | single-transition
stepwise-transition	::=	step-block { step-block }
step-block	::=	**step** step-number: transition-rule
step-number	::=	integer-literal
single-transition	::=	transition-rule
transition-rule	::=	basic-rule | conditional-rule | universal-rule | call-rule | agent-rule
basic-rule	::=	update-rule | block-rule | abbreviation-rule | empty-rule
conditional-rule	::=	if-rule | case-rule | choose-rule | with-rule
universal-rule	::=	forall-rule
agent-rule	::=	create-rule | dispatch-rule | stop-rule | return-rule | destroy-rule
empty-rule	::=	

Entre cada passo da iteração na execução da regra principal de um módulo, os mecanismos de troca de mensagens entre agentes são processados normalmente, independentemente se a regra é de um único ou múltiplos passos.

2.26.1 Atualização de Funções

A regra de atualização é formada por um identificador, que deve ser um nome de função declarada como dinâmica, seguido de seus argumentos, os quais são usados para a determinação do endereço da atualização, e uma expressão, cujo valor é atribuído ao endereço formado. Funções estáticas, derivadas e externas não podem ser atualizadas.

Sintaxe:

update-rule	::=	location := expression
location	::=	dynamic-function-designator [(arguments)]
dynamic-function-designator	::=	function-designator
arguments	::=	expression { , expression }

Por exemplo, uma regra de atualização pode ser da forma $f(x_1, \cdots, x_n) := y$, onde f é uma função dinâmica com assinatura $f : T_1 \times \cdots \times T_n \to T$, cada x_i é uma expressão cujo tipo é T_i, $1 \leq i \leq n$, e y é uma expressão cujo tipo é T.

2.26.2 Bloco de Regras

A regra bloco é composta por uma seqüência de regras, separadas por ponto-e-vírgula, que devem ser executadas simultaneamente. Sua sintaxe é:
block-rule ::= transition-rule { ; transition-rule }

2.26.3 Abreviatura de Termos

A regra *let* é uma regra especial, que não é de transição, e tem a forma:

let x_1 = e_1;
let x_2 = e_2;
...
let x_k = e_k;

onde, para $1 \leq i \leq k$, x_i são identificadores e e_i são expressões. Seu efeito é avaliar cada uma das expressões e_i e continuar a execução da regra de transição em um ambiente onde cada x_i está imediatamente associado à respectiva expressão e_i. No início da próxima iteração da regra de transição todos os valores das variáveis dos *let* são indefinidas, i.e., as associações criadas pela *let* somente tem validade durante uma iteração. Os identificadores definidos em uma regra *let* são meta-variáveis que representam *right values*, não podendo ser, por isto, alvos de atribuições.

A regra *let* tem a seguinte sintaxe:
abbreviation-rule ::= **let** variable-name = expression

Por exemplo, o trecho de código

```
...
let a = head(z);
let b = head(tail(z));
let c = tail(tail(z));
x := a :: b :: c ;
z := c;
...
```

declara os identificadores `a`, `b` e `c`, cujo escopo é toda a regra a partir do ponto de definição do **let**.

2.26.4 Regras Condicionais

Uma regra condicional tem o efeito de executar uma regra escolhida a partir dos valores das guardas de diversas alternativas possíveis. As regras condicionais são: *if, case, with* e *choose*.

Regra if

A regra *if* tem a forma

if g_1 **then** R_1 **elseif** g_2 **then** R_2 $\cdots$ **elseif** g_k **then** R_k **else** R_{k+1} **end**,

onde g_i, $1 \leq i \leq k$, são expressões booleanas denominadas guardas e R_i, $1 \leq i \leq (k+1)$, são regras de transição. Seu efeito é avaliar as guardas g_i na ordem em que aparecem e, quando alguma delas for verdadeira, a regra correspondente é executada e o restante da regra é ignorado. Se nenhuma das guardas for verdadeira, então a regra R_{k+1} é executada.

Sintaxe:

if-rule	::=	**if** expression **then** transition-rule else-part **end**
else-part	::=	{ elsif-term } [**else** transition-rule]
elseif-term	::=	**elseif** expression **then** transition-rule

Regra case

A regra *case* tem a forma:

case e
  of e_1 `=>` R_1`;` **of** e_2 `=>` R_2`;` $\cdots$ **of** e_k `=>` R_k`;`
  otherwise `=>` R_{k+1}`;`
end

onde e é uma expressão de algum tipo discreto (**Bool**, **Char**, **Int** ou alguma enumeração), e_i, $1 \leq i \leq k$, são expressões constantes ou manifestas do mesmo tipo de e, e R_i, $1 \leq i \leq (k+1)$, são regras de transição. Seu efeito é avaliar a expressão e e executar a primeira regra R_i para a qual $e_i = e$. Se para todo valor de i, $e_i \neq e$, então executa-se R_{k+1}. Por exemplo:

```
case head(s) * head(tail(s))
  of 1 => x := 11
  of 3 => x := 17
  of 4 => x := 19
  otherwise => x := 23
end
```

A sintaxe da regra **case** é dada por:

case-rule	::=	**case** expression { case-rule-atom } [case-rule-otherwise] **end**
case-rule-atom	::=	**of** case-expression => transition-rule
case-rule-otherwise	::=	**otherwise** => transition-rule
case-expression	::=	expression

Regra *with*

Uma regra *with* é uma forma especial de *case*, em que, em vez de compararmos o valor de uma expressão com expressões manifestas, comparamos o nome do tipo de uma expressão com os nomes de tipos relacionados nas cláusulas da regra *with* ou com o padrão de estruração do valor da expressão.

A regra *with* permite inspecionar o tipo de elementos de uniões disjuntas e com eles operar, conforme seu tipo, ou então decompor os elementos de um valor composto, do tipo lista, tupla ou nodo de árvore.

A regra *with* tem a forma:

with e
  as $x_1 : T_1$ => $R_1(x_1)$;
  ...
  as $[a\ \ b\ \ c\ \ \cdots]$ => $R_2(a, b, c)$;
  ...
  as $(a :: c)$ => $R_3(a, c)$;
  ...
  as $(a, b, c, \cdots)$ => $R_4(a, c)$;
  otherwise => R_5;
end

onde e é uma expressão cujo tipo normalmente é de uma união disjunta, x_i, $1 \leq i \leq k$, são identificadores, T_i, $1 \leq i \leq k$, são nomes de tipos, e R_i, $1 \leq i \leq (k+1)$, são regras de transição. Seu efeito é avaliar a expressão e e executar a primeira regra R_i em que tipo de e seja T_i. O ambiente em que R_i é executado contém o identificador x_i associado ao valor de e. Se para todo valor de i, o tipo de e for diferente de T_i, então executa-se R_{k+1}.

A sintaxe para esta regra é a seguinte:

with-rule	::=	**with** expression { with-rule-atom } [with-rule-otherwise] **end**
with-rule-atom	::=	**as** with-pattern => transition-rule
with-pattern	::=	[function-name :] pattern
with-rule-otherwise	::=	**otherwise** => transition-rule

Os identificadores usados nos padrões que ocorrem nas cláusulas de um *with* devem ser previamente declarados, entretanto, a associação de cada um deles com a sub-estrutura correspondente do argumento da regra *with* tem escopo restrito à regra de transição vinculada à respectiva cláusula.

Regra *choose*

A regra *choose* tem a forma: **choose** $e_1, e_2, ..., e_k$ **satisfying** `g` **do** `regra` **end**, onde as expressões $e_1, e_2, ..., e_k$ e `regra` são da mesma forma que na regra *forall* e g é uma expressão booleana. O seu efeito é escolher não-deteministicamente elementos dos domínios dados que satisfaçam a guarda `g` e executar a regra dada, associando os nomes de identificadores em $e_1, e_2, ..., e_k$ aos elementos escolhidos.

Sintaxe:

choose-rule	::=	**choose** choose-elements [**satisfying** expression] **do** transition-rule **end**
choose-elements	::=	choose-element { , choose-element }
choose-element	::=	[variable-name :] choose-domain
choose-domain	::=	choose-expression
choose-expression	::=	expression

Uma *choose-expression* é uma expressão de tipo coleção de valores, ou seja, um conjunto ou uma lista.

2.26.5 Regra de Universalização

A regra *forall* tem a forma:

forall $x_1 : U_1$ **,** $\cdots$**,** $x_k : U_k$ **do** R **end**

onde x_i são identificadores, U_i são expressões que denotam coleção de valores, isto é, conjuntos ou listas, e R, uma regra de transição. Seu efeito é criar diversas instâncias da regra `R`, uma para cada valor possível de $e_1, e_2, ..., e_k$ nos domínios dados e executá-las paralelamente. Cada instância é executada em um ambiente em que os identificadores x_i estão associados aos valores atribuídos na instância da regra.

Por exemplo, considere o trecho de código abaixo:

```
f(Int,Int):Int;
transition:
  forall x:1..3,y:1..3 do
    f(x,y) := x + y
  end
  forall a:  agent of M do
    dispatch a;
  end
end
```

Este código é equivalente ao código abaixo:

```
f(Int,Int):Int;
transition:
```

```
    f(1,1) := 2; f(1,2) := 3; f(1,3) := 4; f(2,1) := 3;
    f(2,2) := 4; f(2,3) := 5; f(3,1) := 4; f(3,2) := 5;
    f(3,3) := 6;
  end
```

Identificadores definidos na regra *forall* são meta-variáveis e não podem ser lados esquerdos de atualizações.

A sintaxe da regra *forall* é:

forall-rule	::=	**forall** for-elements **do** transition-rule **end**
for-elements	::=	for-element { , for-element }
for-element	::=	variable-name : for-domain
for-domain	::=	for-expression
for-expression	::=	expression

Uma `for-expression` é uma expressão do tipo coleção de valores (conjuntos ou lista).

2.26.6 Administração de Agentes

Para administrar a execução dos agentes da máquina abstrata, são utilizadas as regras *stop*, *return*, *create*, *dispatch* e *destroy*.

Sintaxe:

create-rule	::=	**create** agent-designator-list
agent-designator-list	::=	agent-designator { , agent-designator }
agent-designator	::=	single-agent-designator | multiple-agent-designator
single-agent-designator	::=	agent-name
multiple-agent-designator	::=	agent-name number-of-agents
number-of-agents	::=	(integer-expression)
integer-expression	::=	expression
dispatch-rule	::=	**dispatch** agent-designator-list
stop-rule	::=	**stop**
return-rule	::=	**return**
destroy-rule	::=	**destroy** agent-designator-list
agent-designator-list	::=	agent-designator { , agent-designator }

A regra *create* tem a forma: **create** x_1**,** $\cdots$**,** x_k, onde x_i são identificadores declarado como sendo do tipo **agent of** M, onde M é o nome do módulo associado a cada um dos agentes. Seu efeito é criar k agentes, cada um devendo executar repetidamente a regra de transição do módulo M_i.

A regra *create* pode criar múltiplos agentes de um mesmo módulo, como ocorre na regra **create** `m(10)`, onde `m:`**agent of** `M`, a qual cria 10 agentes de tipo `M`. A identificação `m` dos agentes criados é um mapeamento do tipo **Int-> agent of** `M`, com domínio no intervalo de 1

ao número de agentes criados, o qual é dado pelo atributo implícito `numberOfAgents`. Assim, no exemplo dado, `m.numberOfAgents` retorna o valor 10. A regra *create* não inicia a execução do agente, a qual deve ser feita explicitamente via a regra *dispatch*.

A regra *stop* termina a execução do agente que está executando no momento, assim que o passo atual for terminado. A execução do agente chega ao fim após o disparo das atualizações do passo.

A regra *return* é semelhante à regra *stop*. Ela só pode aparecer dentro do corpo de uma abstração de regra, e seu objetivo é terminar a execução da ação, voltando o controle ao para o ponto de chamada.

A regra *destroy* tem a forma **destroy** a, onde a é uma expressão de tipo agente. Seu efeito é avaliar a expressão a, obtendo a referência para um agente e terminar a execução deste agente. Se o agente a ser destruído for o agente que está executando no momento, então a sua execução será terminada após os disparos das atualizações do passo, da mesma forma que na execução da regra *stop*. O código abaixo encerra a execução de todos os agentes que executam a regra do módulo `X` e encerra a sua própria execução.

```
forall x : X.agents do destroy x; end;
destroy self;
```

Note que **destroy self** não é equivalente a **stop**, porque após o **destroy** a execução do agente nunca poderá ser retomada. Após a execução de um **stop** a estrutura do agente que o executou continua alocada, de forma que o agente pode voltar a execução se seu estado sofre alguma alteração.

Todo agente tem um atributo implícito `state`, do tipo **State** que indica o estado corrente do agente. Por exemplo, `a.state` informa o estado corrente do agente `a`. Os estado possíveis sao: (i) **new**- estado imediatamente após a criação; (ii) **running**- estado de uma agente em execução; (iii) **blocked**- estado de um agente em execução que se encontra bloqueado, entre uma transição e a próxima, e.g., à espera de uma resposta a pedido enviado a um outro agente; (iv) **dead**- estado de um agente que foi destruido ou executou um **stop**.

2.26.7 Chamada de Abstrações

Uma regra de tipo *chamada de abstração de regra* gera um conjunto de atualizações, que é incluído no conjunto de atualizações existente no ponto de chamada.

Os argumentos de uma abstração que por ela podem ser atualizados devem ser uma função dinâmica ou uma expressão que avalie em algum endereço para atualização. Durante a execução do corpo da abstração, o parâmetro formal compartilhará o endereço do respectivo argumento. No ponto de chamada, as atualizações feitas dentro da abstração só serão percebidas após o disparo de suas atualizações, que ocorrerá ao fim do passo em que a chamada

for feita. Os tipos dos parâmetros formais da abstração de regra devem ser compatíveis com os tipos dos respectivos argumentos.

A execução de abstração do tipo iteração consiste na execução repetida de seu corpo até que seja encontrada a regra *return*. Já a execução de uma ação consiste em executar seu corpo uma única vez, retornando-se imediatamente ao ponto de chamada.

A sintaxe das chamadas é:

call-rule	::=	rule-designator [(rule-arguments)]
rule-arguments	::=	expression { , expression }

Considere o módulo `M` abaixo que define a iteração `mult`:

```
module M
  algebra:
    x, y, z :  Int;
  abstractions:
    action mult(a:Int, b:Int) loop:
      if a < b then a := a + 1 else return end
    end;
  initial state: x := 1; y := 5;
  transition:
    if y <= 10 then y := y + 5; mult(x,y); z := z + y + x
    else stop
    end
end
```

Na definição da iteração `mult`, há dois parâmetros, `a` e `b`. Como o parâmetro `a` é atualizado dentro da abstração, então ele é um parâmetro de saída. Observe que esta iteração incrementa o valor de `a` sucessivamente até que o seu valor seja igual a `b`. Quando isto ocorrer, o comando volta ao ponto de chamada.

Inicialmente, `x` possui o valor 1, `y` possui o valor 5 e `z` possui valor 0. No primeiro passo da execução da regra de transição, `y` tem seu valor alterado para 10 e a iteração `mult` é chamada com os argumentos `x` associado ao parâmetro de saída `a` e 5 associado ao parâmetro `b`. A execução da iteração faz com que o valor de `x` seja sucessivamente incrementado até que seja igual a 5. Após o retorno da abstração, `z` recebe a soma dos valores de `x`, `y` e `z` do início do passo, isto é, 6. No próximo passo, teremos então `x`, `y` e `z` com os valores 5, 10 e 6 respectivamente. No segundo passo, a iteração é novamente chamada associando `x` ao parâmetro `a` e o valor 10 (o novo valor de `y`) ao parâmetro `b`. Novamente, o valor de `x` será incrementado sucessivamente, até que seu valor seja igual a 10, e então a abstração retornará. Após o retorno, `z` receberá a soma dos valores de `x`, `y` e `z` do início do passo, isto é, 21. No terceiro passo, a condição da regra de transição do módulo não é mais satisfeita, então a regra *stop* é executada, e a execução do agente fica suspensa.

2.27 Invariante da Execução

Em um módulo, pode-se definir uma condição que deve ser satisfeita no início e no fim de todo passo de execução da sua regra de transição de estado. Chama-se esta condição de *invariante*.

Na seção de definição de invariantes, **invariant:** *expressão-booleana* especifica uma condição booleana que deve ser satisfeita antes e após cada passo de execução da regra de transição da módulo. Execuções que não satisfazem o invariante em algum momento são consideradas execuções inválidas do módulo. Isto auxilia a prova de propriedades do sistema especificado, tomando-se somente execuções válidas da máquina. Por exemplo, o módulo abaixo define como invariante da execução que `x` deve ser sempre maior que 0 e menor que 100.

```
module M
  algebra:
    x :  Int:= 1;
    external f(x:Int):Int;
  transition:
    if x <= f(x) then x := x + 1
    else stop
    end
  invariant: x > 0 and x < 100
end
```

Se em algum passo, o valor da função externa `f` for tal que faça o valor de `x` ficar fora do limite especificado, então a execução agente é encerrada com erro. O invariante do módulo é verificado sempre antes e depois da execução de sua regra de transição por algum agente.

2.28 Expressões

2.28.1 Introdução

As expressões de Machĭna são as seguintes: expressões sobre os tipos básicos, listas, tuplas, conversão de tipos e uma combinação de operadores e expressões mais simples.

Os operadores de Machĭna são mostrados na tabela a seguir. A ordem de apresentação mostra a precedência dos operadores, sendo que os de precedência mais alta estão na primeira linha e os de precedência mais baixa estão no fim.

old	valor da função operando antes da execução da ação	maior prioridade
- not	inversão de sinal ou de valor lógico	
* / %	multiplicação, divisão, resto	
+ -	soma ou subtração	
= != < > <= >=	operadores de relação	
and	conjunção lógica	
or xor	disjunção inclusiva ou exclusiva	
..	constrói conjunto por intervalo	
is	inspeção de tipo	menor prioridade

Sintaxe:

expression	::=	literal | unop expression | expression binop expression
	|	agregate-expression | expression **is** pattern
	|	function-call | type-identification (expression)
	|	conditional-expression | variable-designator
	|	exists-expression | all-expression
	|	**default** (type-identification) | (expression)
unop	::=	$-$ | **not**| **old**
binop	::=	$*$ | / | % | + | $-$ | = | ! = | < | > | <= | >= | :: | ..
	|	**and** | **or** | **xor** | **in**
function-arguments	::=	expression { , expression }
function-call	::=	function-designator [(function-arguments)]

O operador unário **old**, que fornece o valor do operando antes da execução a ação. Seu operando deve ser o ponto de uma função, i.e., uma função com seus argumentos.

2.28.2 Aplicação de Funções

A aplicação de função tem a forma geral *nome de função*(*argumentos*), onde *argumentos* é uma lista de expressões, cujo comprimento é igual à aridade da função da aplicação. Se a aridade da função for igual a zero, então não se utilizam os parênteses, somente o nome da função. Os tipos dos argumentos devem ser equivalentes aos tipos esperados como parâmetros.

Em uma chamada de função da forma $f(e_1, \cdots, e_n)$, f deve ter sido declarada como uma função de domínio $T_1 \times T_2 \times \cdots \times T_n \rightarrow T$ e o tipo de cada e_i é compatível com T_i, para $1 \leq i \leq n$. O tipo da expressão resultante será T.

2.28.3 Agregados

Os agregados são listas, tuplas, conjuntos ou nodos.

agregate-expression	::=	tuple-expression | list-expression | set-expression
	|	node-expression
tuple-expression	::=	([expression { , expression }])
list-expression	::=	([expression { , expression }])
set-expression	::=	"{"[expression { , expression }] "}"
node-expression	::=	"["{ node-element } "]"
node-element	::=	string-literal | node-expression | variable-designator
	|	function-call
agent-set-expression	::=	**agent of** module-name

Os agregados do tipo lista, conjunto denotam coleção de elementos de mesmo tipo.

Exemplos:

```
type T = tuple(a:Int, b:Int);
type L = list of Int;
type S = set of Int;
x :T; y :L; z :S; w :N; u :  S
...
x := (1,2);
y := (1,2,3,4,5,6);
z := {1,2,3,4,5,6};
u := 1 ..  6;
```

2.28.4 Padrões

Padrões definem a estrutura de listas, tuplas, conjuntos ou nodos. Eles permite que se verifique a estrutura de valores e estabeleça associações de seus constituintes a funções, permitindo-lhe acesso seletivo a estrutura.

pattern	::=	type-identification | tuple-pattern | list-pattern | node-pattern
tuple-pattern	::=	([tuple-pattern-elements])
tuple-pat-elements	::=	tuple-pattern-element { , tuple-pattern-element }
tuple-pattern-element	::=	[function-name :] pattern-element
pattern-element	::=	pattern
list-pattern	::=	([list-pattern-elements])
list-pattern-elements	::=	{ function-name , } function-name :: function-name
node-pattern	::=	"["{ node-pattern-element } "]"
node-pattern-element	::=	literal-string | function-name

Exemplos:

```
type n = ["while" E "do" C];
w :N;
...
w := ["while" e "do" c];
```

2.28.5 Inspeção de Tipos

O operador binário **is** é um operador de inspeção de tipos e de casamento de padrões. No caso de inspeção de tipo, os argumentos de **is** são uma expressão e um nome de tipo, e a operação **is** retorna verdadeiro se o nome do tipo da expressão for igual ao do tipo dado. No caso de casamento de padrão, a estrutura interna da valor da expressão é comparado com o padrão especificado. Um exemplo da utilização do operador **is** está no código a seguir:

```
type X = Bool | Int | Real| (a:Int, b:Real;
x :  X;
...
if x is Int then ...  elseif x is (Int,Int)...  else ...  end
```

2.28.6 Expressões Condicionais

As expressões condicionais podem ser de três tipos, conforme mostra as seguintes regras gramaticais:

conditional-expression ::= if-expression | case-expression | with-expression

Expressão if

Na expressão **if**, as condições indicadas devem ser expressões booleanas e as expressões que ocorrem nas partes **then** e **else** devem ter tipos estruturalmente equivalentes. A expressãi **if** tem tipo equivalente ao tipo destas expressões. Os operandos da expressão **if** são avaliados em sequência à medida que são necessários para se determinar o valor da expressão.

if-expression ::= **if** expression **then** expression else-exp-part **end**
else-exp-part ::= { elsif-exp-part } **else** expression
elsif-exp-part ::= **elseif** expression **then** expression

Expressão case

O argumento de uma expressão **case**, isto é, a expressão após a palavra **case**, pode ser de qualquer tipo que tenha a operação de comparação por igual. A cláusula **otherwise** é obrigatória se as cláusulas da expressão não cobrirem todas as possibilidades de valores do argumento. As expressões que antecedem as setas (`=>`) devem ter tipo equivalente ao tipo do argumento. As expressões após as setas (`=>`) e a da cláusula **otherwise** devem ter tipos compatíveis entre si. As expressões de cada cláusula da expressão **case** são avaliadas em sequência à medida que são necessárias para determinação do valor da expressão.

case-expression	::=	**case** expression { case-exp-clause } [default-case-exp] **end**
case-exp-clause	::=	**of** expression => expression
default-case-exp	::=	**otherwise** => expression

Expressão with

O argumento da expressão **with**, isto é, a expressão que ocoore logo após a palavra **with**, pode ser de qualquer tipo. A cláusula **otherwise** é obrigatória quando as cláusulas da expressão **with** não cobrirem todas as possibilidades de tipo do argumento. Os termos que antecedem as setas (=>) têm o formato de uma declaração de função sem argumento. As expressões após as setas (=>) e a da cláusula **otherwise** devem ter tipos compatíveis entre si. Na avaliação da expressão **with**, 0 tipo do argumento é testado em sequência contra o tipo indicado em cada uma das cláusulas até que se encontre o primeiro tipo que seja, por equivalência nominal, uma projeção do tipo do argumento. Isto ocorrendo, o valor do argumento é associado à variável da cláusula, e a expressão que segue a respectiva seta (=>) dá o valor da expressão **with**. Caso contrário, o valor é dado pela expressão da cláusula **otherwise**.

Sintaxe:

with-expression	::=	**with** expression { with-exp-clause } [default-with-exp] **end**
with-exp-clause	::=	**as** with-exp-pattern => expression
with-exp-pattern	::=	[function-name:] pattern
default-with-exp	::=	**otherwise** => expression

2.28.7 Expressões exist

A expressão **exist** permite determinar se uma coleção contém valores que satisfazem uma dada condição.

exists-expression	::=	**exist** exists-elements **satisfying** expression
exists-elements	::=	exists-element { , exists-element }
exists-element	::=	variable-name : exists-domain
exists-domain	::=	expression

2.28.8 Expressões all

A expressão **all** informa se todos os elementos de uma coleção de valores satisfazem uma dada condição.

all-expression	::=	**all** all-variables **satisfying** expression
all-variables	::=	all-variable { , all-variable }
all-variable	::=	variable-name : var-domain
var-domain	::=	expression

2.28.9 Expressões Especiais

A expressão **default** retorna o valor *default* do tipo esperado. Para cada tipo existe um valor *default* pré-definido. É da forma `default`(T),onde T é um nome de tipo.

A expressão **self** retorna a identificação do agente que estiver executando a regra. Abaixo, seguem alguns exemplos:

```
algebra:
  type T = Int default 100;
  static a :  T := default(T); // a := 100
  static b :  Bool:= default(Bool); // b := false
transition:
  hungry(self) := true, // Atualiza as funções locais ao agente
  thinking(self) := false
```

Capítulo 3

Exemplos de Programas em Machĭna

3.1 Pesquisa Binária

```
module PesquisaBinaria
algebra:
    static n : Int := 100;
    external key : Int;
    i, k, pos, inf, sup, s : Int;
    found : Bool;
    array : Int -> Int;
    f: Input;
initial state:
    open(f,"myvector.dat",s);
    k := key; pos := (n+1)/2; inf := 1; sup := n;
    found := false; i := 1;
transition:
    step 1: readInt(f,array(i), s); i := i + 1;
    step 2: if i <= n then next := 1; end ;
    step 3: let middle = (inf+sup)/2;
            if not found and inf < sup then
                    if array(middle) = k then found := true; pos := middle
                    elseif array(middle) < k then inf := middle + 1 ;
                    else sup := middle - 1;
                    end ;
                    next := 3;
            end
end PesquisaBinaria
```

3.2 Ordenação por Seleção

```
module Selecao
algebra:
    static n : Int;
    type Index = Int;
    type Array = Index -> Int;
    external n : Int;
    external array: Array;
    i, k, j, r: Int;
    f: Array;
    g: Input;
initial state:
    open(g,"myvector.dat",s);
    f := array; n := length(array);
    i := 1; k := 1; j := 1; r := 1;
transition:
    step 1: readInt(g,array(r), s); r := r + 1;
    step 2: if r <= n then next := 1; end;

    step 3: if i < n then
            k := i; j := i + 1
            end
    step 4: j := j + 1;
            if j <= n then
                 if f(j) < f(k) then
                       k := j; next := 4
                 end
            end
    step 5: i := i + 1; next := 3;
            if k != i then
                f(k) := f(i); f(i) := f(k)
            end

end Selecao
```

3.3 Números Primos

```
module Primos
algebra:
    type  Int;
    primo : Int -> Bool;
initial state:
    forall n:2..1000 do primo(n) := true end
transition:
    forall num1:2..1000, num2:2..1000 do
       if num2 < num1 and num1%num2 = 0 then
           primo(num1) := false
       end
    end
end Primos
```

3.4 Especificação da Semântica de *Tiny*

Tiny é uma linguagem imperativa de pequeno porte que contém somente comandos e expressões. A definição de *Tiny* será feita em cinco módulos:

1. Módulo principal (`MainProgram`) – contém a regra principal da especificação e as rotinas de inicialização;
2. Módulo de Expressões (`Expressions`) – contém as regras de avaliação de expressões;
3. Módulo de Comandos (`Commands`) – contém as regras de execução de comandos;
4. Módulo de Operações (`Operations`) – contém as regras de operação na pilha.
5. Módulo de Globais (`Globals`) – contém as declarações dos principais tipos e funções da especificação de *Tiny*.

3.4.1 Módulo de Globais (`Globals`)

```
module Globals
algebra:
    public type Undefined = public enum {UNDEFINED};
    public type Id = String;
    public type Program = List;
    public type InputFile = list of Int;
    public type OutputFile = list of Int;
    public type Value = Bool | Int;
    public type Memory = Id -> Value | Undefined default UNDEFINED;
    public type KeyWord =  public enum {
          TADD, TASSIGN, TCOND, TEXCHANGE, TEQUALS, TNOT, TOUTPUT, TREAD, TWHILE
    };
    public type Opstack = List;
    external p : Program;
    external i : InputFile;
    public program : Program := p;
    public infile  : List := i;
    public outfile : OutputFile := nil;
    public opstack : Opstack := nil;
    public memory  : Memory ;
    public error   : Bool := false ;
end Globals
```

3.4.2 Módulo de Operações (Operations)

```
module Operations
include Globals(program,outfile,infile,opstack,error,memory);
abstractions:
    public action treatOutput is
        if head(program) is x:Int then
             outfile := concat(outfile,[x]);
             program := tail(program); opstack := tail(opstack)
        else error := true
        end
    end treatOutput

    public action treatAssign  is
        opstack := tail(tail(opstack));
        memory(String(head(tail(opstack)))) := Globals.Value(head(program))
    end treatAssign

    public action treatExchange is
        opstack := tail(opstack);
        program := head(tail(program))::head(program)::tail(tail(program))
    end treatExchange

    public action treatAdd is
        let x = head(program);
        let y = head(tail(program));
        if x is Int and y is Int then
                 program := (Int(x) + Int(y))::tail(tail(program));
                 opstack := tail(opstack)
        else error := true
        end
    end treatAdd

    public action treatNot is
        let b = head(program);
        if b is Bool then
             program := (not Bool(b))::tail(program);
             opstack := tail(opstack)
        else error := true
        end
    end treatNot
```

```
    public action treatCond is
        if head(program) is Bool then
            if head(program) is Bool then
                 program := head(tail(program))::tail(tail(tail(program)))
            else program := tail(tail(program))
            end;
            opstack := tail(opstack);
        else error := true
        end
    end treatCond

    public action treatEquals is
        let x = head(program);
        let y = head(tail(program));
        if (x is Int) and (y is Int) then
              program := (Int(x) = Int(y))::tail(tail(program));
              opstack := tail(opstack)
        else error := true;
        end
    end treatEquals

    public action treatValue is
        if head(opstack) is Globals.KeyWord then
            case Globals.KeyWord(head(opstack))
                 of Globals.TOUTPUT   => treatOutput;
                 of Globals.TASSIGN   => treatAssign;
                 of Globals.TEXCHANGE => treatExchange;
                 of Globals.TADD      => treatAdd;
                 of Globals.TNOT      => treatNot;
                 of Globals.TCOND     => treatCond;
                 of Globals.EQUALS    => treatEquals ;
             end
        else error := true
        end
    end treatValue

end Operations
```

3.4.3 Módulo de Expressões (Expressions)

```
module Expressions
include Globals(program,outfile,infile,opstack,error,memory);
abstractions:
    public action readMemory is
        let id = String(head(program));
        if memory(id) = UNDEFINED then
              error := true
        else program := memory(s) :: tail(program)
        end
    end

    public action treatNot is
        program := tail(program);
        opstack := Globals.TNOT::opstack
    end

    public action treatEquals is
        program := tail(program);
        opstack := Globals.TEQUALS::opstack
    end

    public action treatAdd is
        program := tail(program);
        opstack := Globals.TEXCHANGE::Globals.TADD::opstack
    end;

    public action treatRead is
        if infile = nil then
             error := true
        else
             program := head(infile)::tail(program);
             infile := tail(infile)
        end
    end
  end Expressions
```

3.4.4 Módulo de Comandos (Commands)

```
module Commands
include Globals(program,outfile,infile,opstack,error,memory);
abstractions:
    public action  treatAssign is
        program := tail(tail(program));
        opstack := Globals.TASSIGN::head(tail(program))::opstack
    end;
    public action treatOutput is
        program := tail(program);
        opstack := Globals.TOUTPUT::opstack
    end;
    public action treatConditional is
        program := tail(program);
        opstack := Globals.TCOND::opstack
    end
    public action treatWhile is
        let second    = head(tail(program));
        let third     = head(tail(tail(program)));
        let whileExp  = second;
        let whileCom  = Globals.TWHILE::second::third::nil;
        let whileTrue = third::whileCom::nil;
        let whileCont = tail(tail(tail(program)));
        program := whileExp::whileTrue::nil::whileCont;
        opstack := Globals.tcond::opstack
    end
  end Commands
```

3.4.5 Módulo Principal (MainProgram)

```
module MainProgram
include Commands, Expressions, Operations,
        Globals(program,outfile,infile,opstack,error,memory);
abstractions:
    public action flattenProgram is
         program := concat((list(head(program)), tail(program))
    end
transition:
    if program != nil and not error then
       with head(program)
          as t:List   => flatProgram;
          as s:String => Expressions.readMemory;
          as p:Globals.KeyWord =>
             case p
                of Globals.TNOT    => Expressions.treatNot;
                of Globals.TEQUALS => Expressions.treatEquals;
                of Globals.TADD    => Expressions.treatAdd;
                of Globals.TREAD   => Expressions.treatRead;
                of Globals.TASSIGN => Commands.treatAssign;
                of Globals.TOUTPUT => Commands.treatOutput;
                of Globals.TCOND   => Commands.treatConditional;
                of Globals.TWHILE  => Commands.treatWhile;
             end;
          as x:Globals.Value => Operations.treatValue
       end
    end
end MainProgram

machina Tiny
   agent of MainProgram
end Tiny
```

3.5 Jantar dos Filósofos

```
machina DiningPhilosophers
    agent of Host;
end DinningPhilosopers
interface Host
    type ForkId;
    action getForks(in leftFork:ForkId, in rightFork:ForkId, out granted:Bool);
    action freeForks(in leftFork:ForkId, in rightFork:ForkId);
end Host
module Host
import Philosopher(setRightFork,setLeftFork);
algebra:
    public type ForkId = Int;
    fork: ForkId -> Bool;
    p, t, first : agent of Philosopher;
    agentCount : Int;
    external numberOfGuests: Int;
    philosophers : agent of Philosopher;
abstractions:
    public action getForks(leftFork:ForkId, rightFork:ForkId, granted:Bool) is
        if fork(rightFork) or fork(leftfork) then
             granted := false;
        else granted := true;
             fork(rightFork) := true; fork(leftFork) := true;
        end

    end getForks

    public action freeForks(leftFork:ForkId, rightFork:ForkId) is
        fork(rightFork) := false; fork(leftfork) := false;
    end freeForks
initial state:
    create philosophers(numberOfGuests);
transition:
   step 1: p := philosophers(1); first := p;
           agentCount := 2; forkId := 2;
   step 2: if agentCount <= philosophers.numberOfAgents then
           t := philosophers(agentCount); agentCount := agentCount + 1;
           else next := 4;
           end
   step 3: fork(forkId) := false;
           p.setRightFork(forkId); t.setLeftFork(forkId); p := t;
           forkId := forkId + 1; next := 2;
   step 4: fork(1) := false; p.setRightFork(1); first.setLeftFork(1);
   step 5: forall p: philosophers do dispatch p; end
```

```
end Host
```

```
interface Philosopher
    action setRightFork(in f: ForkId);
    action setLeftFork(in f: ForkId);
end Philosopher

module Philosopher
import Host(ForkId,getForks,freeForks);
algebra:
    leftFork, rightFork: ForkId;
    thinking, hungry, eating : agent of Philosopher -> Bool;
    host: agent of Host;
    granted: Bool;

abstractions:
    public action setRighFork(r: ForkId) i
          rightFork := f;
    end setRight

    public action setLeftFork(f: ForkId) is
          leftFork := f;
    end setLeft
initial state:
    host := Host.theAgent;
    thinking(self) := true;
transition:
    step 1: if thinking(self) then
                  thinking(self) := false; hungry(self) := true; next := 1;
            elseif hungry(self) then
                  host.getForks(leftFork,rightFork,granted);
            else  eating(self) := false;
                  thinking(self) := true;
                  host.freeForks(leftFork,rightFork); next := 1;
            end;
    step 2: if granted then
                  eating(self) := true;
                  hungry(self) := false;
            end;
            next :=1;
end Philosopher
```

3.6 Semáforo

```
module Semaforo
algebra:
    reqs   : Agent -> Bool;
    time   : Agent -> Int;
    chosen : Agent;
    clock  : Int;

abstractions:
    public action lock() is
    loop:
        if not reqs(self) then
            reqs(self) := true; time(self) := clock
        end;
        if chosen = self then return end
    end

    public action unlock() is
    loop:
        if chosen = self then
            chosen := undef; clock := clock + 1
        end
    end
initial state:
    clock := 0;
transition:
    if chosen = undef and not isEmpty(reqs) then
        choose x:reqs satisfying time(x) < clock do
            chosen := x; reqs(x) := false
        end
    else clock := clock + 1
    end
end
```

3.7 Tipo Abstrato de Dados Pilha

```
module EdfStack
algebra:
    public type Stack(T) = tuple(elements:Int -> T, top:Int);
abstractions:
    public action initStack(s:Stack(T)) is
        s.top := 0;
    end initStack;

    public action push(s:Stack(T), x:T) is
        s.elements(s.top + 1) := x; s.top := s.top + 1;
    end push

    public action pop(s: Stack(T), x:T, error:Bool) is
         if s.top > 0 then
               x := s.elements(s.top); s.top := s.top - 1;
               error := false;
         else error := true;
         end
    end push

    public action empty(s:Stack(T), isempty:Bool) is
         isempty := s.top = 0;
    end empty
end EdfStack

module TestStk
include EdfStack(initStack,Stack,push,pop,empty);
algebra:
   s1:Stack(Int);  s2:Stack(Char);
   a,b:Int; c,d:Char;
   errorIns1, errorIns2, s1Empty, s2Empty :Bool;
initial state:
   initStack(s1);
   initStack(s2);
transition:
   step 1: a := 10; c := 'a';
   step 2: push(s1,a); push(s2,c);
   step 3: pop(s1,b,errorIns1); pop(s2,d,errorIns2);
   step 4: empty(s1,s1Empty);empty(s2,s2Empty);
end TestStk
```

Capítulo 4

Avaliação da Metodologia

4.1 Precisão

Toda especificação deve descrever, de forma precisa, o sistema real correspondente. Pode-se utilizar a metodologia de especificação para descrever um sistema via uma sintaxe particular e uma semântica associada clara e bem definida. Se a semântica da metodologia de especificação não é clara, descrições que utilizam a metodologia podem não ser mais claras do que o sistema original que está sendo descrito. ASM utiliza estruturas matemáticas clássicas (conjuntos, sistemas de transição), que são modelos precisos e bem conhecidos, para descrever estados da computação [22, 23].

4.2 Demonstração de Correção da Especificação

Diversos aspectos da metodologia tornam fácil demonstrar propriedades da especificação ou a sua correção em relação ao sistema real especificado em ASM.

Uma técnica utilizada para especificar sistemas é a técnica de refinamento passo a passo, sugerido em [23] e utilizado em [25, 28]. Esta técnica permite que o projetista especifique um modelo do sistema em alto nível e detalhe passo a passo os diversos objetos e operações do sistema, como na metodologia *top-down*. Em [9], esta técnica foi aliada à verificação formal. A verificação consistiu em provar que o modelo em mais alto nível (o "*ground model*") estava correto em relação ao mundo real. Em seguida, para cada passo do refinamento, provou-se que a introdução de novos detalhes preservava a correção em relação ao modelo imediatamente anterior.

A demonstração de propriedades do sistema de transição também é simples, pelos seguintes motivos:

- a transição é óbvia, visto que não há noção de fluxo de controle como nas linguagens de programação, onde as possíveis seqüências de execução são dinâmica e imprevisíveis;
- não há efeitos colaterais na execução de regras, o que significa que, com uma simples inspeção das regras, determina-se o conjunto de atualizações.

Além disso, podemos verificar informalmente a correção da especificação, por inspeção das regras, como sugerido em [23], ou executando-se a especificação.

4.3 Generalidade

A metodologia ASM é útil em uma grande variedade de domínios: sistemas seqüenciais [11, 35]; sistemas paralelos e distribuídos [2, 9]; sistemas de tempo real [19, 27]. Em [10], há muitos outros exemplos de aplicações de ASM em diversos tipos de sistemas dinâmicos discretos. Muitas metodologias de especificação são, na prática, úteis apenas em domínios particulares. Por exemplo, a Semântica Denotacional é aplicável a sistemas seqüenciais, como algumas linguagens de programação. Entretanto, a sua adaptação para computação distribuída é problemática. A extensão da metodologia ASM para computação distribuída é natural, como pode ser visto em [23].

4.4 Facilidade de Aprendizado

ASM utiliza somente notação matemática simples e bem conhecida e uma sintaxe semelhante à de linguagens de programação imperativa populares. Pode-se trabalhar com a metodologia, utilizando qualquer conhecimento ou técnica existente, evitando a camisa de força dos métodos formais. Pode-se construir um sistema real seguindo um caminho natural de explicação, utilizando notação matemática padrão, de maneira que o modelo ASM resultante torna-se simples, transparente e fácil de ler, entender e manipular. Isto refuta, para a metodologia ASM, uma objeção que freqüentemente é feita contra a utilização de métodos formais para sistemas grandes e complexos. Diz-se que o programador médio não tem base matemática suficiente para ser capaz de aplicar métodos formais. No caso de ASM, apenas experiência com algoritmos é suficiente para desenvolver ou compreender as especificações e as provas.

4.5 Facilidade de Leitura e de Escrita

A leitura e a escrita de toda especificação devem ser fáceis. Se a notação é difícil de ler e escrever, poucas pessoas irão utilizá-la. Programas em ASM utilizam uma sintaxe bastante simples, na forma de pseudocódigo, que pode ser lida inclusive por novatos. O único conhecimento necessário para compreender uma especificação ASM é o entendimento de programação.

Outros métodos de especificação, em especial a Semântica Denotacional [20], utilizam conceitos menos usuais, que podem tornar a leitura e a escrita tarefas muito complicadas.

Qualquer programador é capaz de ler e escrever uma especificação ASM, pois os conceitos utilizados são simples. Ao contrário de Semântica Denotacional, que utiliza uma meta-linguagem baseada no lambda-cálculo, ou da Semântica Operacional Estruturada [33], que utiliza conceitos de programação em lógica, especificações em ASM utilizam uma linguagem na forma de pseudo-código semelhante às linguagens de programação do paradigma imperativo, que é mais comum ao programador médio.

ASM permite o uso de termos e conceitos do domínio do problema, com um mínimo de codificação notacional. Muitos métodos populares de especificação formal requerem uma grande quantidade de codificação notacional, o que pode tornar a tarefa de especificação muito difícil.

4.6 Escalabilidade

Muitos métodos formais tradicionais não são escaláveis. Eles funcionam muito bem para exemplos pequenos, freqüentemente inventados para ilustrar o método. Entretanto, quando utilizados em grandes sistemas reais, muitos deles caem em uma explosão combinatorial ou simplesmente falham. A comunidade ASM trabalha na definição de linguagens do mundo real: em [4, 5, 6], Egon Börger explica toda a linguagem Prolog, e, em [25], é apresentada a definição da linguagem C.

O uso de ASM permite lidar com a complexidade de sistemas reais, através da construção de hierarquias de níveis de sistemas. Com múltiplas camadas, pode-se facilmente examinar características particulares do sistema ignorando outras. A demonstração de propriedades do sistema também é mais simples se utilizarmos esta técnica. Em [7], Börger argumenta que a metodologia é escalável para especificação de sistemas integrados de *hardware/software*.

4.7 Possibilidade de Execução

Além da inspeção direta, outra maneira de testar a correção de uma especificação é executá-la diretamente. Métodos como VDM [31] ou Z [30, 34] não são diretamente executáveis.

As principais vantagens em executar uma especificação são:

1. permite "experimentar"a especificação antes que o algoritmo seja implementado ou quando não há implementações do algoritmo disponíveis;
2. várias condições do ambiente externo podem ser representadas por parâmetros na especificação, o que significa que há mais oportunidades de inserção de situações de falha;

3. permite combinar teste com verificação formal.

4.8 Ambientes de Execução de ASM

Existem diversas implementações de ASM, entre interpretadores e compiladores. Dentre os interpretadores, os mais importantes são a implementação de Michigan[15] e o ASM-Workbench[16]. Nestes, o foco principal não está na eficiência da execução, portanto técnicas robustas de otimização não são aplicadas. Entre os compiladores, os que aplicam técnicas de otimização são o XASM[1], e o EVADE[37]. No compilador XASM, a única otimização feita é a representação eficiente de funções por meio de tabelas *hash*. A única otimização realizada pelo compilador da ferramenta EVADE é a eliminação de subexpressões comuns. Nenhuma das ferramentas existentes aplicam técnicas de otimização que ataquem diretamente os recursos de elevado custo do modelo ASM, visto que utilizam somente técnicas clássicas para otimização de código de linguagens imperativas.

Neste relatório apresentamos a linguagem chamada Machĭna, que implementa os principais recursos de ASM e algumas extensões.

Bibliografia

[1] M. Anlauff, P. Kutter, and A. Pierantonio. Tool Support for Language Design and Prototyping with Montages. In *Proceedings of Compiler Construction (CC'99)*. Springer Lecture Notes in Computer Science, 1999.

[2] D. Bèauquier and A. Slissenko. On Semantics of Algorithms with Continuous Time. Technical Report 97-15, Dept. of Informatics, Université Paris-12, October 1997.

[3] D. Bèauquier and A. Slissenko. The Railroad Crossing Problem: Towards Semantics of Timed Algorithms and their Model-Checking in High-Level Languages. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *LNCS*, pages 201–212. Springer, 1997.

[4] E. Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36–64. Springer, 1990.

[5] E. Börger. A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer, 1990.

[6] E. Börger. A Logical Operational Semantics for Full Prolog. Part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output. In Y. Moschovakis, editor, *Logic From Computer Science*, volume 21 of *Berkeley Mathematical Sciences Research Institute Publications*, pages 17–50. Springer, 1992.

[7] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer, 1995.

[8] E. Börger and U. Glässer. Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In Y. Gurevich and E. Börger, editors, *Evolving Algebras – Mini-Course, BRICS Technical Report (BRICS-NS-95-4)*, pages 128–153. University of Aarhus, Denmark, July 1995.

[9] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[10] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998.

[11] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1997.

[12] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[13] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim and J. Gruska and J. Zlatuska, editor, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic*, number 1450 in LNCS. Springer, August 1998.

[14] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[15] G. Del Castillo, I. Durdanovic, and U. Glässer. The Evolving Algebra Interpreter Version 2.0. pages 191–214, 1996.

[16] G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

[17] G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra Abstract Machine. In H. Kleine Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 191–214. Springer, 1996.

[18] A. Gill. *Applied Algebra for the Computer Sciences.* Prentice Hall, Englewood Cliffs, 1976.

[19] U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[20] M J C Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[21] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[22] Y. Gurevich. Evolving Algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 423–427, Elsevier, Amsterdam, the Netherlands, 1994.

[23] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[24] Y. Gurevich. The Sequential ASM Thesis. Technical Report MSR-TR-99-09, Microsoft Research, February 1999.

[25] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[26] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of *LNCS*, pages 266–290. Springer, 1996.

[27] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 295–328. Oxford University Press, 1995.

[28] J. Huggins. Kermit: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 247–293. Oxford University Press, 1995.

[29] J. Huggins and R. Mani. Evolving Algebras Interpreter v. 2.0. Technical report, 1992.

[30] D. C. Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* Oxford University Press, 1988. ISBN 0-19-859667-7.

[31] C. B. Jones. *Systematic Software Development using VDM (second edition).* Prentice Hall, 1990.

[32] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Computer Science. Prentice-Hall, 1987.

[33] F. Nielson and H. R. Neilson. *Semantics with Applications - A Formal Introduction.* John Wiley and Sons, New York, N.Y., 1992.

[34] J. M. Spivey. *The Z Notation.* Prentice Hall, New York, 1989.

[35] K. Stroetmann. The Constrained Shortest Path Problem: A Case Study In Using ASMs. *Journal of Universal Computer Science*, 3(4):304–319, 1997.

[36] F. Tirelo, M. A. Maia, R. S. Bigonha, and V. O. Di Iorio. Tutorial Sobre Máquinas de Estado Abstratas. In *Anexo dos Proceedings of the III Simpósio Brasileiro de Linguagens de Programação*, pages 1–30, Porto Alegre, RS, Maio 1999. SBC.

[37] J. Visser. Evolving Algebras. Master's thesis, Delft University of Technology, 1996.

[38] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[39] N. Wirth. What Can We Do About the Unnecessary Diversity of Notation for Syntatic Definition? *Communication of the ACM*, 20(11):822–823, 1977.