

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Laboratório de Linguagens de Programação

Projeto Machina

por

**Roberto da Silva Bigonha
Fábio Tirelo
Marcelo de Almeida Maia
Marcelo de Silva
Marco Túlio de Oliveira Valente
Mariza Andrade da Silva Bigonha
Vladimir Oliveira Di Ioro
RT 009/99**

Caixa Postal, 702
30.161-970 – Belo Horizonte
Minas Gerais – Brasil
julho de 1999

Sumário

1	Normae Formalis Ex Machīna	1
2	Máquinas de Estados Abstratas	4
2.1	Introdução	4
2.2	O Modelo Formal de ASM	5
3	Linguagem de Programação Algébrica	9
3.1	Introdução	9
3.2	Ferramentas Existentes	10
3.3	A Ferramenta Proposta	11
3.4	Conclusão	11
4	Semântica Formal Padrão em ASM	12
4.1	Introdução	12
4.2	Semântica Padrão	12
4.3	Semântica de Linguagens Utilizando ASM	13
4.4	Trabalho Proposto	14
5	Avaliação Parcial de Máquinas de Estado Abstratas	15
5.1	Introdução	15
5.2	Avaliação Parcial	16
5.3	Conclusões	18
6	Especificação de Ambientes para Computação Móvel em ASM	19
7	Especificação Formal da Interação de Componentes de Sistemas Computacionais	22
8	Publicações e Referências	24
8.1	Publicações Recentes da Equipe	24

Capítulo 1

Normae Formalis Ex Machīna

O projeto **Machīna** tem por objetivo o estudo, projeto e desenvolvimento de um novo sistema de programação algébrica e sua aplicação na construção de geradores de compiladores baseados em semântica, na formulação de semântica formal de linguagens de programação e na especificação de ambientes de computação móvel.

Os paradigmas de programação podem ser classificados com base no modelo computacional subjacente. A literatura frequentemente destaca os paradigmas imperativo, funcional lógico e algébrico.

Programação imperativa é baseada na arquitetura de von Neumann, onde os conceitos de variável, atribuição e sequenciamento são elementos básicos para a composição da semântica dos programas. Um programa em linguagem imperativa pode ser considerado uma formalização da solução do problema por ele resolvido, mas os modelos teóricos que podem ser usados para descrever o funcionamento deste tipo de programas são demasiadamente complexos ou limitados, tornando muito difícil provar propriedades de solução descrita no paradigma imperativo.

Na programação funcional o modelo computacional é a teoria matemática das funções recursivas, onde instrumentos como cálculo de ponto fixo, prova por indução, etc, tornam possível e factível raciocinar sobre programas funcionais. Neste paradigma, o próprio programa é uma formalização rigorosa baseada em estruturas matemáticas estabelecidas e poderosas. Por exemplo, a semântica denotacional de uma linguagem de programação é uma descrição formal que, em última instância, é um programa em λ -cálculo.

A programação lógica, baseada na formulação de fatos e regras de inferência, tem seu alicerce na lógica matemática, a qual oferece todo o aparato necessário para o estudo e derivação de propriedades da solução especificada pelo programa.

No paradigma algébrico, a teoria matemática subjacente é a álgebra, na qual são definidos vocabulário, universo de valores, interpretação dos símbolos do vocabulário e, opcionalmente, axiomas e teoremas. A interpretação dos elementos do vocabulário pode ser apresentada de várias formas, como por exemplo, via axiomas e asserções do cálculo de predicados ou por meio do λ -cálculo.

Em alguns sistemas [1, 2], programação algébrica é entendida como uma programação

baseada em *reescrita*. Nestes sistemas, reescrita de um símbolo por sua interpretação provê um método para executar uma especificação algébrica. Por exemplo, na bem conhecida definição recursiva do n -ésimo número de Fibonacci, têm-se as seguintes relações:

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

Este sistema pode ser visto como um sistema de regras de reescrita, que podem ser usadas para computar $F(n)$. De um ponto de vista mais geral, pode-se dizer que a aplicação de uma operação aritmética a constantes é uma reescrita baseada na interpretação da operação.

No Projeto **Machĩna**, consideram-se algébricos os programas escritos com base na descrição de um sistema algébrico, com uma definição de vocabulário, do universo de valores, da interpretação de símbolos do vocabulário e de uma regra de transição, pela qual estabelece-se a norma para redefinição da interpretação de elementos do vocabulário da álgebra. Em essência, a regra de transição implementa uma operação de reescrita de interpretação.

O modelo formal em que o estilo de programação algébrico que propomos se baseia são as *Evolving Algebras* ou Máquinas de Estados Abstratas (ASM) de Gurevich [29]. ASM é um formalismo baseado no conceito de sistemas dinâmicos, onde os estados são modelados por meio de álgebras, e a regra de transição define as mudanças de estado. Desta forma, projeto **Machĩna** visa o desenvolvimento de um sistema de Programação Algébrica com o objetivo de fazer valer a adaptação: *Normae Formalis Ex Machĩna*.

ASM já foi utilizada para descrever formalmente a semântica operacional de diversas linguagens, como, por exemplo, a linguagem Java e sua compilação para JVM [40], a linguagem C [47], Prolog [22, 23, 24], Modula-2 [35] e Occam [26].

O uso de ASM apresenta diversas vantagens, dentre as quais, podemos citar:

- precisão, graças a uma semântica simples, baseada em modelos matemáticos simples e bem conhecidos, como por exemplo conjuntos e sistemas de transição;
- demonstração de propriedades, graças à facilidade em se identificar os passos do algoritmo e à ausência de efeitos colaterais na execução de regras;
- generalidade, pois é aplicável a uma grande variedade de domínios: sistemas seqüenciais, paralelos, distribuídos, de tempo real;
- facilidade de aprendizado, por utilizar notação matemática simples e bem conhecida e uma sintaxe semelhante a de linguagens do paradigma imperativo;
- facilidade de leitura e escrita, graças à notação na forma de pseudo-código, que é de conhecimento geral; outros métodos, como a semântica denotacional, utilizam conceitos menos usuais, o que pode tornar a leitura e a escrita tarefas muito complicadas;

- possibilidade de execução, o que permite *experimental* a especificação, antes da implementação do algoritmo, testar várias condições do ambiente externo e combinar testes com verificação formal.

A equipe do projeto é constituída de:

- Fábio Tirelo: Mestrando
- Marco Túlio de Oliveira Valente: Doutorando.
Professor da Pontifícia Universidade Católica de Minas Gerais.
- Marcelo de Almeida Maia: Doutorando, com defesa prevista para agosto de 1999.
Professor da Universidade Federal de Ouro Preto, MG.
- Marcelo Luiz Silva: Doutorando.
Professor da Universidade Federal de Ouro Preto, MG.
- Nahur Moraes da Fonseca: aluno de Iniciação Científica.
- Vladimir Oliveira Di Iorio: Doutorando.
Professor da Universidade Federal de Viçosa, MG.
- Mariza Andrade da Silva Bigonha: Pesquisadora
Professora Adjunta da Universidade Federal de Minas Gerais.
- Roberto da Silva Bigonha: Coordenador do projeto, pesquisador.
Professor Titular da UFMG.

No próximo capítulo, os conceitos de ASM são apresentados visando fornecer subsídios sobre o modelo formal subjacente ao Projeto **Machina**.

Nos capítulos subsequentes, apresentamos um resumo de cada um dos subprojetos que serão desenvolvidos, a saber:

1. Linguagem de Programação Algébrica
2. Semântica Formal Padrão em ASM
3. Avaliação Parcial em ASM
4. Especificação de Ambientes de Computação Móvel em ASM
5. Especificação Formal da Interação de Componentes de Sistemas Computacionais

Capítulo 2

Máquinas de Estados Abstratas

2.1 Introdução

Com o objetivo de prover um contexto para o detalhamento do projeto **Machina**, apresentamos a seguir uma pequena introdução ao modelo formal das Máquinas de Estados Abstratas.

ASM é um modelo formal de especificação de algoritmos, baseado no conceito de estado, descrito por uma álgebra, e de transição de estados. A descrição do estado inicial da máquina é então um sistema algébrico consistindo em vocabulário, universo de valores e interpretação inicial dos símbolos do vocabulário.

As mudanças de estado na máquina são dadas por uma regra de transição cuja função é modificar as interpretações de símbolos do vocabulário da álgebra, dando origem a um novo estado, que é então descrito por uma nova álgebra.

Uma ASM $\mathcal{A}$ é uma coleção finita de nomes de funções, cada uma com uma aridade fixa. Um *estado* de $\mathcal{A}$ é um conjunto não vazio, o superuniverso, junto com interpretações dos nomes da assinatura de funções sobre os elementos do superuniverso. As interpretações são designadas *funções básicas* do estado. As funções básicas podem ser alteradas à medida que $\mathcal{A}$ muda de estado, mas o superuniverso se mantém inalterado.

Formalmente, supondo um superuniverso X , uma função básica de aridade r é uma função $X^r \rightarrow X$. Quando $r = 0$, a função é designada *elemento distinto*. O superuniverso sempre contém os elementos distintos **true**, **false** e **undef**, definidos como *constantes lógicas*. O elemento **undef** é utilizado para representar funções parciais, por exemplo, $f(\vec{a}) = \text{undef}$ significa que f é indefinida para a tupla $\vec{a}$. Uma relação r -ária sobre X pode ser vista como uma função $X^r \rightarrow \{\text{true}, \text{false}\}$. Um *universo* U é um tipo especial de função básica: uma relação unária geralmente identificada pelo conjunto dos elementos x tais que $U(x) = \text{true}$, i.e., $\{x : U(x)\}$.

Um programa de $\mathcal{A}$ é uma regra de transição, que pode, em sua forma mais simples, ser *instrução de atualização*, *construtor de bloco* ou *construtor condicional*.

Uma instrução de atualização tem o formato $f(\vec{t}) := t_0$, onde f é o nome de uma função

da assinatura de $\mathcal{A}$, $\bar{t}$ é uma tupla de termos cujo tamanho é igual à aridade de f , e t_0 é outro termo. Os termos não possuem variáveis livres e são construídos recursivamente usando-se nomes de elementos distintos e aplicação do nome de uma função a outros termos. De maneira informal, a semântica da atualização acima é a seguinte: a tupla $\bar{t}$ é avaliada, e o valor da função básica f aplicada à tupla é alterado para o valor da avaliação de t_0 . Ou seja, o nome f passa a ter uma nova interpretação.

Um construtor condicional tem genericamente a forma:

```
if g0 then R0
elseif g1 then R1
...
elseif gk then Rk
endif
```

A semântica do comando condicional é a seguinte: a regra R_i , $0 \leq i \leq k$, é executada se os termos booleanos $g_0, \dots, g_{i-1}$ são avaliados para **false**, e g_i é avaliado para **true**.

Um construtor de bloco é um conjunto de regras. Sua semântica é a seguinte: todas as possíveis atualizações das regras contidas no bloco são disparadas em paralelo. Se uma atualização contradiz outra, uma escolha não-determinista é realizada.

Uma execução de um programa de $\mathcal{A}$ é uma seqüência de estados, onde o estado seguinte é obtido a partir do anterior através da execução da regra de transição do programa. A maioria das implementações de ASM determina o final da execução quando o disparo da regra de transição não produz nenhuma atualização. Por razões de eficiência, em **Machina**, o fim de uma execução é obtido pela execução da função primitiva **stop**.

2.2 O Modelo Formal de ASM

Definição 1 Vocabulário:

Um *vocabulário* Υ é uma coleção finita de nomes de funções e relações, 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.

Definição 2 Estado:

Um *estado* S é uma álgebra, 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 outros conjuntos;

- uma função $Val : \Upsilon \rightarrow X^* \rightarrow X$, que fornece a interpretação dos nomes de funções pertencentes a Υ como funções de $X^* \rightarrow X$.

Para os conjuntos do superuniverso, estão definidas as operações usuais como, no caso dos inteiros, adição, multiplicação, etc.

A noção de estado é importante, pois cada estado constitui um passo na execução da máquina.

Definição 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 alguns pontos. Caso contrário, dizemos que a função é *estática*.

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

Definição 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 $\bar{x} = (x_1, \dots, x_r)$, então $f(\bar{x})$ é um termo.

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

- 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 $(x_1, \dots, x_r)$ é uma tupla de comprimento r , então

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

Estas regras são as mais simples de ASM. Existem outros tipos de regras mais poderosas (vide [29]).

Definição 5 Conjunto de Atualização:

Atualização de um estado é um par (loc, v) , onde loc é uma localização, isto é, um ponto de uma função dado seu argumento, e v é o valor que a função passará a ter neste ponto.

O conjunto de atualizações de um estado S por uma regra R , $Updates(R, S)$, é definido como o conjunto de pares (loc, v) determinados pela regra R .

Definição 6 Disparo:

O disparo de uma regra R em um estado S produz um novo estado S' , tal que, se o conjunto $Updates(R, S)$ for consistente, então

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

Se o conjunto $Updates(R, S)$ não for consistente, então o efeito do disparo de R em S pode ser nulo, isto é, o estado S' resultante é 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 $Fire(R, S)$ como o estado resultante do disparo de R em S .

Definição 7 Especificação ASM:

Uma *especificação ASM* é uma tupla 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 outros 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 $X^r \rightarrow X$. 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 por meio de qualquer formalismo existente, em particular, via regras de transição do modelo ASM. Pode-se utilizar também mecanismos como λ -cálculo, funções recursivas, entre outros.

Definição 8 Função Externas:

Na maioria dos sistemas dinâmicos discretos, a execução pode ser influenciada pelo ambiente. Intuitivamente, um ambiente ativo é um agente externo. Desta maneira, utilizamos *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, utilizamos funções externas para descrever implicitamente o não-determinismo; há outras construções para especificarmos explicitamente o não-determinismo.

Definição 9 Execução de uma Especificação ASM:

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

Capítulo 3

Linguagem de Programação Algébrica

3.1 Introdução

Este trabalho consiste no projeto, definição e implementação de um ambiente de programação algébrica baseado em ASM, que seja fácil de utilizar e eficiente na execução das especificações.

Para isso, desenvolveremos uma nova linguagem de programação, que denominamos **Machĩna**, um compilador da linguagem e um ambiente de depuração da especificação.

Machĩna é uma linguagem baseada em ASM que contém construções de alto nível, tais como tipos, abstrações de regras e suporte à modularidade.

Machĩna é fortemente tipada e possui a facilidade da inferência de tipos. O sistema de tipos de **Machĩna** é baseado no sistema de tipos de SCRIPT [20].

Um exemplo de programa em **Machĩna** são os módulos mostrados abaixo:

```
module M1
  import: M(u);
  algebra:
    sort F is (Int, List of Int) -> Int;
    public Z is List of Int;
    static f(x;z) = x + head(z);
    g(x,z):F = x - head(z);
    dynamic x : Int;
  init:
    x := 100;
  transition:
    if x < 10
      then Modulo2.t(x);
      else u(x,[[1,2],[3],[4]]);
```



```

    end,
    x := x + 1;
end

module M2
  algebra:
    type X is Int;
    dynamic public h : Int -> Int;
    action public t(x) =
      if x = 0
        then h(x) := 10;
        else h(x) := 100;
      end
    end;
    action public u(x ; y:List) =
      let k = head(y)
      in with k do
        p : Int -> h(x) := p;
        q : List of Int -> h(x) := head(q);
        else h(x) := -1;
      end
    end
  end;
end
end

```

3.2 Ferramentas Existentes

A literatura descreve duas importantes implementações de ASM. A primeira é um interpretador desenvolvido, em 1996, na Universidade de Michigan [44]. Esta implementação possui diversos problemas, entre eles, temos que o ambiente é de difícil utilização e a linguagem utilizada para escrever especificações é muito restrita, não permitindo construções de alto nível, tipos e escrita de especificações modulares. A segunda implementação é um ambiente desenvolvido na Universidade de Paderborn, em 1998 [45]. Esta implementação possui um ambiente de fácil utilização e sua linguagem possui algumas construções de alto nível, como por exemplo, definição de tipos. O maior problema desta implementação é que o próprio ambiente é interpretado, o que torna a interpretação do programa muito ineficiente.

3.3 A Ferramenta Proposta

Estamos no momento especificando a linguagem **Machĩna** que tem suporte à modularidade e que permite construções de alto nível. **Machĩna** é uma linguagem fortemente tipada, com a facilidade de inferência de tipos em tempo de compilação.

Programas em **Machĩna** serão compilados para gerar código C otimizado. Com a compilação e otimização para C, pretendemos obter uma execução mais eficiente da especificação.

Além destes recursos, o sistema que estamos implementando possuirá uma interface para depuração da especificação e possibilitará a inserção de novos módulos, como, por exemplo, verificadores formais. Para a implementação de tal sistema, os problemas que tentaremos resolver consistem em definir e desenvolver:

- Mecanismos para representação de funções dinâmicas, de modo a permitir eficiência na recuperação de valores e atualização de estados;
- Mecanismos eficientes para avaliação das guardas e identificação de ponto fixo;
- Mecanismos para gerência de espaço de nomes;
- Mecanismos para armazenamento e recuperação de dados estruturados, tais como listas, tuplas e mapeamentos;
- Definição da arquitetura do modelo de regras de transição;
- Mecanismos de implementação e representação de funções externas.

3.4 Conclusão

As contribuições deste trabalho serão as seguintes:

- Definição da linguagem **Machĩna**, que é uma linguagem com suporte a módulos e diversas construções de alto nível, baseada em ASM.
- Implementação de **Machĩna**.
- Geração de código de alta qualidade para execução.
- Ambiente de execução, que permite acompanhamento passo a passo da execução e inserção de novos módulos.

Capítulo 4

Semântica Formal Padrão em ASM

4.1 Introdução

O objetivo principal do trabalho é obter um padrão para definição da semântica das linguagens de programação utilizando as Máquinas de Estado Abstratas. Observa-se atualmente uma falta de padrão, com cada pesquisador utilizando um estilo particular. Espera-se que comparações entre linguagens, melhor compreensão, facilidades na definições de especificações, provas de propriedades sobre programas, sejam auxiliadas pela padronização.

4.2 Semântica Padrão

Michael J. C. Gordon propôs em [25] um padronização para semântica denotacional. Nesta abordagem, a semântica dos construtos de uma linguagem é provida por meio de denotações definidas por funções que mapeiam objetos sintáticos em objetos semânticos. Um exemplo de utilização do padrão proposto por Gordon foi a definição da semântica da linguagem Prolog [36]. Gordon procurou definir conceitos matemáticos comuns às linguagens a serem modeladas. Desta forma, foi estabelecido um estilo de definição semântica composto por módulos comuns. Na especificação semântica encontram-se:

- a sintaxe abstrata da linguagem: composta pelas cláusulas sintáticas, domínios sintáticos e as metavaráveis que varrem estes domínios. Aqui tem-se a formalização da sintaxe da linguagem com os aspectos relevantes à semântica;
- os domínios semânticos: como exemplos tem-se *State* para modelar o estado computacional; *Store* que modela a associação de locações com valores que podem ser armazenados, estes últimos pertencentes ao domínio *Storable Values*; *Environment* que modela a ligação entre identificadores e valores que podem ser ligados aos identificadores, estes valores pertencem ao domínio *Denotable Values*; *Expressible Values* que é o domínio dos valores que podem ser resultado de expressões; e os domínios

padrão de continuções que passam um resultado intermediário para um código adequado do programa visando atingir uma resposta final, que é definida no domínio *Final Answer*;

- funções semânticas: são funções que mapeiam um construto da linguagem, pertencente a um domínio sintático, ao seu significado ou denotação, pertencente ao domínio semântico;
- funções semânticas auxiliares que são um recurso de modularização das funções semânticas;
- cláusulas semânticas: são as cláusulas que descrevem como é feito o mapeamento das funções semânticas. Utiliza-se como recurso o lambda cálculo.

Com esta proposta, Gordon estabeleceu um estilo de especificação da semântica das linguagens de programação. Alguns benefícios advindos deste trabalho são: a fácil identificação dos conceitos semânticos de uma linguagem por vários pesquisadores; facilidade de estudo para especialistas e interessados em semântica denotacional; e uma metalinguagem comum para especificação da semântica utilizada em publicações científicas.

4.3 Semântica de Linguagens Utilizando ASM

As Máquinas de Estado Abstratas, introduzidas por Yuri Gurevich [29], são um conceito rigoroso, preciso e não ambíguo, que permite a modelagem matemática de um algoritmo, em particular, a semântica de uma linguagem de programação.

A meta principal da metodologia ASM é prover especificações formais sucintas e executáveis para algoritmos. Aqui, o termo algoritmo é genérico, podendo incluir linguagens de programação, arquiteturas, protocolos distribuídos e de tempo real, especificação formal da interação de componentes de sistemas computacionais [33], conexão de sistemas baseadas em ASM [34], geração automática de compiladores [30], etc. No caso de linguagens de programação, uma linguagem pode ser vista como um algoritmo, um algoritmo universal que toma um dado programa como parte de seus dados [Gur93a]. Como exemplo pode-se citar formalização de Smalltalk [21], Prolog [22, 23, 24], C [47], Occam [26], Modula-2 [35], C++ [39], Java [40], etc. ASM permite que as especificações sejam feitas no nível de abstração desejado, tornando o algoritmo o mais próximo possível da realidade, no seu próprio nível de abstração. A semântica de uma linguagem é provida por meio da abordagem operacional. Desta forma, os significados dos construtos são definidos através das transições de estados. Por exemplo, o algoritmo que define a semântica começa em um estado inicial S_0 , e torna-se S_1 , S_2 , ..., etc. A transformação dos estados é definida através de regras de transições, onde uma quantidade de tarefas bem definidas é realizada em cada passo. Os estados são as álgebras, sendo compostos por funções e objetos matemáticos que modelam os significados.

A especificação semântica é provida por vários níveis de abstração, álgebras, para uma mesma linguagem. Isto é útil, pois pode-se examinar a semântica de características particulares de uma linguagem de programação no nível de abstração desejado, facilitando a compreensão. Por exemplo, uma linguagem do paradigma seqüencial pode ser especificada pelas seguintes álgebras: comandos, expressões, inicialização e alocação de memória, e chamada e retorno de funções [Gur 93b].

Até o momento, foram construídas várias aplicações de ASM para modelagem da semântica de linguagens de programação. Observa-se que não há uma padronização definida, bem como um estilo comum de especificação.

4.4 Trabalho Proposto

Serão desenvolvidas pesquisas para desenvolver: uma metodologia universal de formulação da semântica de linguagens de programação baseada em ASM; uma metodologia de estruturação de definições semânticas em ASM em níveis de abstração convenientes; uma padronização da semântica de linguagens de programação em ASM; uma métrica para comparações de linguagens de programação baseada no modelo formal; e uma técnica para validar a metodologia proposta.

O trabalho a ser desenvolvido pretende propor uma padronização para a semântica, definindo de forma padrão os conceitos das ASM, como funções, domínios, vocabulários comuns a todas semânticas. Será necessária uma abordagem que defina estilo de organização dos módulos de uma máquina abstrata, provendo recursos para modelagem de construtos que permitam computações seqüenciais, definições de tipos abstratos de dados, de classe, de polimorfismo, herança, e outros conceitos existentes nas linguagens atuais. O estilo de especificação a ser proposto será baseada na linguagem **Machina**. Isto permitirá que a especificação seja executável, susceptível a depurações, e permitirá comparações com outras especificações. Além disto, pode-se provar propriedades sobre programas, permitindo verificar se dois códigos diferentes resultam em um mesmo significado desejado.

Capítulo 5

Avaliação Parcial de Máquinas de Estado Abstratas

5.1 Introdução

A semântica de uma linguagem de programação L pode ser descrita nesse modelo especificando-se um interpretador para a linguagem L . Um *avaliador parcial* é um algoritmo que, recebendo um programa e parte de seus dados de entrada, produz um novo programa designado por residual ou especializado. O programa residual, se executado com o restante da entrada, gera os mesmos resultados que o programa original, se executado com a entrada completa.

Neste trabalho, procuramos desenvolver técnicas para construir um avaliador parcial para a linguagem das Máquinas de Estado Abstratas. A compilação de um programa escrito em uma linguagem L para ASM pode ser realizada por meio da avaliação parcial de um interpretador de L . Um compilador de L para ASM pode ser automaticamente gerado via auto-aplicação do avaliador parcial.

As ASMs podem ser usadas na descrição da semântica de linguagens de programação, arquiteturas de sistemas, protocolos distribuídos e de tempo real, etc. Usando esse modelo, uma forma de se especificar a semântica de uma linguagem de programação L é escrever um interpretador Int para L . O interpretador recebe um programa S , escrito em L , e seus dados de entrada x , simulando a execução de S sobre x .

Dado um programa P com entradas x e y , a *avaliação parcial* de P em relação a x é um novo programa P_x que recebe apenas uma entrada y e produz os mesmos resultados que P , se lhe fossem fornecidos x e y . As entradas x e y são designadas, respectivamente, *estática* e *dinâmica*. O avaliador parcial realiza uma mistura de execução com geração de código, motivo pelo qual o processo foi designado “*mixed computation*” e o avaliador comumente chamado de `mix` [54].

Suponha disponível um avaliador parcial `mix`. Suponha também que Int seja um interpretador de uma linguagem L , escrito em uma linguagem L_{Int} , que recebe um programa

S escrito em L e seus dados de entrada x . A avaliação parcial de Int , dado S (estático), resulta em um programa S' , geralmente escrito em L_{Int} , que recebe x e produz a mesma saída de S . Ou seja, S' é o resultado da compilação de S para a linguagem L_{Int} . A avaliação parcial do próprio mix , dado como entrada estática um interpretador Int de L , resulta em um compilador de L para L_{Int} . Finalmente, a avaliação parcial de mix dado como entrada estática o próprio mix resulta em um gerador de geradores de compiladores. A possibilidade da realização desses procedimentos foi descoberta por Futamura [55] no início da década de 70, mas só foram realizados com sucesso pela primeira vez em meados dos anos 80 [59].

Em nosso trabalho, procuramos desenvolver técnicas utilizadas para construir um avaliador parcial, que designaremos por mix_1 , para a linguagem das Máquinas de Estado Abstratas. O avaliador parcial pode ser aplicado a interpretadores de linguagens de programação escritos em ASM, tendo como entrada estática um programa da linguagem interpretada. Como explicado anteriormente, esses interpretadores podem ser vistos como a descrição da semântica de linguagens em ASM. O resultado da avaliação parcial é um programa compilado, sendo assim é realizada uma *compilação dirigida por semântica*.

Apresentamos também uma versão de mix escrita na própria linguagem ASM, que designaremos por mix_2 (a primeira versão é implementada em Java). Isso nos permite a auto-aplicação, ou seja, avaliar parcialmente o próprio mix . Se for fornecida como entrada estática o interpretador de uma linguagem L , pode-se produzir automaticamente um compilador de L para ASM, ou seja, *geração de compiladores dirigida por semântica*. Um avaliador parcial para ASM foi apresentado anteriormente em [57], mas não permitia a auto-aplicação e conseqüentemente não possibilitava a geração de compiladores.

5.2 Avaliação Parcial

Um *avaliador parcial* recebe como entradas um outro programa P e parte da entrada de P , designada por *entrada estática*. Vamos identificar a entrada estática como in_1 , e o novo programa construído como P_{in_1} , designado por residual ou *especializado*. A entrada restante, que identificaremos como in_2 , é designada como *dinâmica*. O programa residual, quando executado com o restante da entrada de P , produz os mesmos resultados que P produziria se fosse executado com ambas as entradas.

Seguindo a notação utilizada em [58], se P é um programa escrito na linguagem L , $\llbracket P \rrbracket_L$ é uma função que denota a sua semântica. Quando não for importante, poderemos omitir o subscrito L da notação. Sendo assim, a definição equacional do avaliador parcial mix é a seguinte:

$$out = \llbracket P \rrbracket_S(in_1, in_2)$$

$$P_{in_1} = \llbracket mix \rrbracket_L(P, in_1)$$

$$out = \llbracket P_{in_1} \rrbracket_T (in_2)$$

As linguagens envolvidas são L , usada para implementar o avaliador parcial mix , S , a linguagem fonte dos programas submetidos ao avaliador parcial e T , a linguagem objeto dos programas especializados produzidos. Geralmente, S e T são idênticas, mas existem casos onde elas são distintas. A avaliação parcial é vantajosa quando in_2 varia mais do que in_1 e P_{in_1} executa mais rápido sobre in_2 do que P , sobre in_1 e in_2 .

Como o avaliador mix é um programa com duas entradas, ele pode servir de entrada para si próprio. Futamura foi o primeiro a sugerir essa abordagem, no que passou a ser conhecido como *três projeções de Futamura* [55]. Supondo int um interpretador de uma linguagem qualquer, escrito em S , a primeira projeção mostra que compilação através de avaliação parcial sempre gera programas corretos:

$$out = \llbracket source \rrbracket (input)$$

$$out = \llbracket int \rrbracket (source, input)$$

$$out = \llbracket \llbracket mix \rrbracket (int, source) \rrbracket (input)$$

$$out = \llbracket target \rrbracket (input)$$

Assim temos $target = \llbracket mix \rrbracket (int, source)$, ou seja, o programa objeto é resultado da avaliação parcial de um interpretador, dado como estático um programa específico. A segunda projeção refere-se à geração de compiladores através de auto-aplicação de mix :

$$target = \llbracket mix \rrbracket (int, source)$$

$$target = \llbracket \llbracket mix \rrbracket (mix, int) \rrbracket (source)$$

$$target = \llbracket compiler \rrbracket (source)$$

Temos então $compiler = \llbracket mix \rrbracket (mix, int)$. Observe que havíamos feito a suposição de que S era a linguagem fonte dos programas submetidos a mix . No caso da auto-aplicação, isso quer dizer que o próprio mix deve ser escrito na linguagem S .

5.3 Conclusões

Os resultados iniciais alcançados foram animadores, principalmente no que se refere à aplicação da primeira projeção de Futamura. Os resultados conseguidos pela compilação via avaliação parcial de um interpretador foram de boa qualidade. Resultados semelhantes estão sendo obtidos com descrições de semântica de linguagens mais complexas, em especial um subconjunto da linguagem Java.

Em relação à geração de compiladores dirigida por semântica, o trabalho está apenas começando. O primeiro passo será melhorar o tamanho do compilador gerado para exemplos simples e implementar em `mix2` as otimizações presentes em `mix1`. Em seguida, compiladores para linguagens mais complexas serão gerados e avaliados.

Outra fonte de trabalhos futuros será a investigação da avaliação parcial de programas envolvendo paralelismo e concorrência, área ainda com poucos resultados efetivos. A utilização da linguagem ASM poderá facilitar esse estudo, pois as extensões *ASM Paralela* [56], *ASM Distribuída* [56] e *ASM Interativa* [61, 60] permitem exprimir esses conceitos de forma bastante simples e direta.

Capítulo 6

Especificação de Ambientes para Computação Móvel em ASM

Na década de 90, assistimos a um crescimento exponencial da Internet e especificamente da WWW, a qual tornou-se uma grande infra-estrutura computacional cobrindo todo o planeta. No entanto, o que se deseja atualmente é ampliar a utilização computacional da *Web*, isto é, desenvolver sistemas distribuídos globais que funcionem tendo como base a infra-estrutura já instalada.

A principal dificuldade para se desenvolver tais sistemas deriva do fato de que a *Web* não é um sistema distribuído tradicional, isto é, os recursos propostos desde a década passada para desenvolvimento de sistemas distribuídos, tais como chamada remota de procedimentos (RPC) e distribuição de objetos, não se aplicam à *Web*. Basicamente, isto se deve à existência de três fenômenos que somente são observáveis na *Web* [3]:

- Localizações Virtuais: correspondem a domínios administrativos criados com objetivo específico de proteger informações e recursos de acessos externos.
- Localizações Físicas: em uma infra-estrutura global como a *Web*, independentemente de avanços nas tecnologias de rede, não se consegue tornar transparente aos sistemas as distâncias físicas existentes entre os diversos domínios.
- Flutuações na Banda de Passagem: decorrem dos congestionamentos e das interrupções de conexão tão comuns entre os diversos domínios da *Web*.

Além destes três fenômenos, existe um quarto que, apesar de ser observável nos sistemas distribuídos projetados para redes locais, *não* é observável na *Web*:

- Falhas: como na *Web* os atrasos tem duração imprevisível, a noção de falha se confunde com a de atrasos longos.

Assim, a questão de como se “programar a WWW” resume-se à questão de como desenvolver programas capazes de detectar e reagir a estes fenômenos. Um estilo de computação

que vêm sendo proposto atualmente com este objetivo é o de computação móvel, o qual corresponde à noção de que os programas podem se mover de forma autônoma de um nodo para outro da rede¹. Neste modelo, ao se mover, o programa leva consigo não apenas o seu código, mas também o estado corrente de sua execução. Sendo um estilo de computação novo, existe todo um esforço atual em formalizá-lo, dando origem a uma área de pesquisa que vêm sendo chamada de Cálculo de Processos Móveis. Dentre os formalismos propostos recentemente com este fim, destaca-se o Cálculo de Ambientes [3], de Luca Cardelli. O Cálculo de Ambientes é um cálculo de processos inspirado no π -cálculo [7] e cuja principal abstração é a noção de ambiente, definido como um local delimitado por uma fronteira em cujo interior acontecem computações. Ambientes são organizados hierarquicamente e podem se mover para dentro ou para fora de outros ambientes, sendo esta mobilidade regulada por capacidades que os processos devem possuir. O conceito de mobilidade é, portanto, associado com a idéia de cruzar as fronteiras de um ambiente. O Cálculo de Ambientes, no entanto, é um formalismo teórico mínimo cujo principal objetivo é capturar as abstrações essenciais de computação móvel, isto é, ele não é adequado à implementação de sistemas móveis de médio ou grande porte. Para isso, torna-se necessário o desenvolvimento de linguagens de programação que incorporem tais abstrações. Ressalte-se que esse fato já ocorreu com outros cálculos de processos, como, por exemplo, com o CSP [5], que deu origem à linguagem Occam [6] e com o π -cálculo, que mais tarde originou a linguagem Pict [8]. O próprio Cardelli reconhece a necessidade de se propor tais linguagens, tendo inclusive cunhado o termo Linguagens de Longa Distância (em inglês, *Wide Area Languages*) para designá-las. Uma linguagem adequada à implementação de sistemas móveis na Internet deverá atender aos seguintes princípios de projeto [3, 10]:

- Não pressupor a existência de estado global: em uma rede com milhões de nós como a Internet, com novos nós sendo interligados a todo instante, é inviável assumir a existência de um estado global compartilhado por todos elementos. Assim, uma linguagem de programação adequada à Internet não deve permitir a construção de programas que partam desta premissa.
- Não pressupor conectividade contínua: em uma rede em que atrasos são comuns e muitas vezes se confundem com falhas é fundamental o uso de uma linguagem de programação que permita o desenvolvimento de programas capazes de se comunicar apenas com um subconjunto de seus nós clientes ou servidores, ou mesmo de forma desconectada.
- Supor a existência de localizações físicas e virtuais: em uma rede de longa distância, a localização de uma computação e dos recursos que a mesma tenha acesso é essencial para sua eficiência. Portanto, a linguagem deve tornar explícito aos programas sua localização corrente.

¹No contexto deste trabalho, o termo computação móvel é usado como tradução do inglês *mobile computation*. Ressalte-se que existe ainda a expressão *mobile computing*, usada para designar a situação em que os elementos da rede é que são móveis, como no caso das redes sem fio.

- Suportar reconfiguração dinâmica: em uma rede em que mudanças físicas de topologia e de nós provedores de serviços e recursos ocorrem com grande frequência é importante dispor de uma linguagem que permita o desenvolvimento de programas que se adaptem a estas constantes mudanças no ambiente.
- Controlar o acesso a recursos: em uma rede aberta como a Internet é imprescindível a existência de alguma política de segurança no acesso aos recursos e serviços providos pelos diversos domínios. Assim, a linguagem deve prover mecanismos que permitam a proteção de recursos compartilhados.

O objetivo deste trabalho é, portanto, investigar como as abstrações fundamentais para computação móvel na *Web* podem ser transportadas para uma linguagem de programação adequada ao desenvolvimento de sistemas neste modelo e introduzi-las na linguagem **Machina**.

Máquinas de Estado Abstratas, particularmente **Machina**, podem ser vista como uma linguagem que possui apenas os recursos matemáticos mínimos necessários à especificação de outras linguagens de programação mais complexas. Este fato presta-se com precisão aos objetivos deste projeto, uma vez que não se está interessado em outros recursos que não aqueles diretamente relacionados com o problema de computação móvel. Além desta vantagem, o uso de ASM também se justifica pelas seguintes observações:

- Em ASM, o conceito de estado encontra-se precisamente definido, o que torna mais natural a especificação do tipo de mobilidade em que estamos interessados, a mobilidade de estado.
- Em ASM, já existe o conceito de ambiente (*environment*), porém com o objetivo de prover funções externas a uma computação. O que deverá ser proposto então no trabalho é uma extensão do conceito de ambiente visando à especificação de computação móvel.

Uma primeira noção do potencial de ASM para modelagem de sistemas móveis é mostrada em [9]. Especificamente neste projeto, pretende-se analisar a introdução em **Machina** dos seguintes conceitos fundamentais a qualquer linguagem de programação para sistemas móveis:

- Primitivas e ambientes para permitir mobilidade de estado;
- Mecanismos para tratamento de exceções em sistemas móveis;
- Sistemas de tipos para regular mobilidade;
- Comunicação e interação entre componentes móveis.

Capítulo 7

Especificação Formal da Interação de Componentes de Sistemas Computacionais

Apesar de uma grande evolução ao longo dos últimos anos e de benefícios significativos já demonstrados, o uso de métodos formais ainda continua pouco aceito em ambientes industriais [12].

Com esta pesquisa, queremos dar uma contribuição no sentido de tornar o uso de métodos formais uma realidade na prática. Acreditamos que o uso efetivo de especificações formais no processo de desenvolvimento de software pode mudar a maneira como os projetistas abordam o desenvolvimento de software, de tal forma a efetivar a construção sistemática de programas corretos.

Nossa proposta é inspirada nas idéias de Peter Wegner[19], que argumenta que o grande problema na especificação de sistemas de computação é a interação entre os componentes do sistema. Segundo ele, algoritmos modelam funções, enquanto sistemas interativos modelam os programas do mundo real. Propomos uma abordagem inovadora para especificar como componentes de um sistema interagem entre si.

A abordagem é inovadora justamente porque dá ênfase à especificação explícita de como os componentes interagem. Nas abordagens tradicionais de especificação, a parte dinâmica da interação é especificada implicitamente no comportamento de cada componente, e a parte estática é especificada por meio das interfaces dos módulos do sistema, com a indicação dos dados e funções que são exportados e importados por um componente.

Como arcabouço para desenvolvimento de nossas idéias utilizaremos o método formal ASM (*Abstract State Machines*)[56], o qual é um sistema de transição de estados. Propomos uma extensão às ASM, a qual chamaremos de IASM (*Interactive Abstract State Machines*)[14].

A extensão é baseada na especificação da interação entre os elementos de computação via primitivas de troca de mensagens e de configuração dinâmica da topologia de interconexão destes elementos, além da composição e sincronização das respectivas primitivas.

Definimos a semântica formal de nossa proposta em um estilo similar ao de Gurevich, quando da formalização das ASM [56]. Além disso, propomos um esquema de tradução de IASM para ASM tipada[17]. A introdução de tipos na especificação tem como objetivo auxiliar no processo de verificação das regras de formação das especificações e tornar o método orientado para ferramentas.

Mostramos que a nossa abordagem pode ser aplicada com sucesso a diferentes classes de problemas como agentes móveis[15, 18], protocolos de comunicação [13], componentes de software [16, 18]. O estudo destes problemas mostra a viabilidade do método e aponta para a necessidade de uma ferramenta de especificação para facilitar o uso da abordagem na especificação de outros sistemas.

O encaminhamento natural da pesquisa nesta etapa é a definição da sintaxe concreta para a linguagem do modelo IASM e a implementação de um ambiente de execução de especificações IASM. A implementação terá como base o compilador de **Machina**[38].

Todos os estudos de casos desenvolvidos manualmente com as IASM poderão ser testados e verificados com a ferramenta desenvolvida.

Capítulo 8

Publicações e Referências

8.1 Publicações Recentes da Equipe

1. Vladimir O. Di Ioro, Roberto S. Bigonha, and Marcelo A. Maia. A self-applicable partial evaluator for asm. In *Proceedings of the Annual ASM User Group Meeting at the World Congress on Formal Methods for the Development of Computing System*, 21 pages, Toulouse, França, September 1999.
2. Marcelo A. Maia, Vladimir O. Di Ioro, and Roberto S. Bigonha. An asm-based approach for connection-oriented systems. In *Proceedings of the Annual ASM User Group Meeting at the World Congress on Formal Methods for the Development of Computing System*, 21 pages, Toulouse, França, September 1999.
3. Marcelo A. Maia and Roberto S. Bigonha. Interaction based-semantics for mobile objects. In *Proceedings of the III Simpósio Brasileiro de Linguagens de Programação*, pages 93–107, Porto Alegre, RS, Maio 1999. Sociedade Brasileira de Computação.
4. Marcelo A. Maia and Roberto S. Bigonha. Rules for verication of interactive systems. In *Proceedings of the III Simpósio Brasileiro de Linguagens de Programação*, pages 207–212, Porto Alegre, RS, Maio 1999. Sociedade Brasileira de Computação.
5. Vladimir O. Di Ioro and Roberto S. Bigonha. Avaliação parcial de máquinas de estado abstratas. In *Proceedings of the III Simpósio Brasileiro de Linguagens de Programação*, pages 207–212, Porto Alegre, RS, Maio 1999. Sociedade Brasileira de Computação.
6. Mariza A. S. Bigonha, AbdelAli Ed-Dbali, Roberto S. Bigonha, P. Ribeiro, Flávia, Pierre Deransart, and José Siqueira. Projection of hyperpro document. In *Proceedings of the III Simpósio Brasileiro de Linguagens de Programação*, pages 171–185, Porto Alegre, RS, Maio 1999. Sociedade Brasileira de Computação.

7. Fábio Tirelo, Marcelo A Maia, Roberto S. Bigonha, and O. Di Ioro, Vladimir. 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. Sociedade Brasileira de Computação.
8. Marco T. O. Valente, R. S. Bigonha, Marcelo A. Maia, and Antônio Aa. F. Loureiro. Especificação de agentes móveis usando máquinas de estado abstratas. In *Proceedings of the I Workshop de Comunicação sem Fio*, 9 páginas, Belo Horizonte, MG, julho 1999. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais.
9. M.A. Maia, V.O Di Ioro, and Roberto S. Bigonha. Interactive abstract state machines. In *Proceedings of the 5th Workshop on Abstract State Machines*, pages 37–49, Magdeburg, Germany, Outubro 1998.
10. Mariza A.S. Bigonha and Roberto S. Bigonha. A generator of code generator for superscalar architectures. In *Proceedings of the XIV Conferencia Latinoamericana de Informatica – CLEI 98*, pages 49–59, Quito, Equator, Outubro 1998. CLEI.
11. Marcelo .A. Maia and Roberto S. Bigonha. An interactive abstract machine language. In *Anais do Workshop Brasileiro de Métodos Formais*, pages 106–110, Porto Alegre, Brasil, Outubro 1998.
12. Flávia P. Ribeiro, Deransart. P., Roberto S. Bigonha, A. Ed-Dbali, M. A. S. Bigonha and J. Siqueira. Um ambiente para desenvolver programação em lógica baseado no paradigma de estilo literário. In *Anais do VII Semana de Iniciação Científica da UFMG*, pages 204–204, Belo Horizonte, Brasil, 26-31 de Outubro 1998. UFMG.
13. M.A. Maia and Roberto S. Bigonha. The formal specification of the interactive abstract state machines language. Technical Report 005/98, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, MG, setembro 1998.
14. M.A. Maia and Roberto S. Bigonha. Objetos concorrentes em máquina de estados abstratos. In *Proceedings of the II Simpósio Brasileiro de Linguagens de Programação*, pages 126 – 138, Belo Horizonte, MG, Setembro 1997. Sociedade Brasileira de Computação.
15. P. Deransart, Roberto S. Bigonha, P. Parot, M.A.S. Bigonha, and Siqueira J. An hypertext based environment to write literate logic programs. In *Proceedings of the JICSLP’96 - Joint International Conference and Symposium on Logic Programming*, pages 246 – 252, Bonn, Germany, September 1996. Association for Logic Programming.

16. P. Deransart, Roberto S. Bigonha, P. Parot, M.A.S. Bigonha, and Siqueira J. A literate logic programming system. In *Proceedings of the I Simpósio Brasileiro de Linguagens de Programação*, pages 1 – 15, Belo Horizonte, MG, September 1996. Sociedade Brasileira de Computação.

Bibliografia

- [1] Letichevsky, A. A. et alii, *Algebraic Programming System APS (user Manual)*, Technical Report Glushkov Institute of Cybernetic, 1999.
- [2] Goguen, J., Grant Malcolm, *Algebraic Semantics of Imperative Programming (Foundations of Computing)*, 1999.
- [3] Cardelli, L. and Gordon, A.D. *Mobile Ambients*. In *Foundation of Software Science and Computational Structures*, Maurice Nival (ed.), Lecture Notes in Computer Science 1378, Springer, 1998.
- [4] Gurevich, Y. *Evolving Algebras 1993: Lipari Guide*. In *Specification and Validation Methods*. E. Börger (ed.), Oxford University Press, 1995.
- [5] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [6] Jones, G. *Programming in Occam*. Prentice-Hall, 1987.
- [7] Milner, R.J., Parrow, J. and Walker, D. *A Calculus of Mobile Processes*. Parts 1-2, *Information and Computation*, 100(1), 1992.
- [8] Pierce, B.C. and Turner, D.N. *Pict: A Programming Language Based on the Pi-Calculus*. Indiana University Technical Report, 1997.
- [9] Valente, M.T.O., Bigonha, R.S., Maia, M.A. e Loureiro, A.A.F. *Especificação de Agentes Móveis em Máquinas de Estado Abstratas*. I Workshop de Comunicação Sem Fio, Julho, 1999.
- [10] Vitek, J, and Castagna, G. *Seal: A Framework for Secure Mobile Computations*. Available in <http://cuiwww.unige.ch/OSG/people/jvitek/Publications/wipl.ps.gz>.
- [11] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide*. In E. Börger, editor, *Specification and Validation Methods*, páginas 9–36. Oxford University Press, 1995.
- [12] J.C. Knight, C.L. DeJong, M.S. Gobble, e L.G. Nakano. Why are formal methods not used more widely? In *Proceedings of the Fourth NASA Langley Formal Methods Workshop*, Hampton, Virginia, September 1997.

- [13] M.A. Maia, V.O. Di Iorio, e R.S. Bigonha. Interacting Abstract State Machines. In *V International Workshop on Abstract State Machines*, Magdeburg, Germany, September 1998.
- [14] M.A. Maia e R.S. Bigonha. The Interactive Abstract State Machine Language. In *I Workshop Brasileiro de Mitodos Formais*, Porto Alegre, Outubro 1998.
- [15] M.A. Maia e R.S. Bigonha. Interaction Based Semantics for Mobile Objects. In *III Simpssio Brasileiro de Linguagens de Programacao*, Porto Alegre, Maio 1999.
- [16] M.A. Maia e R.S. Bigonha. Modelagem Formal de Arquiteturas e Componentes de Software via Máquinas de Estado Interativas. submitted to XIII Simpósio Brasileiro de Engenharia de Software, Florianopolis, 1999.
- [17] M.A. Maia e R.S. Bigonha. Rules for Verification of Interactive Systems. In *III Simpssio Brasileiro de Linguagens de Programacao*, Porto Alegre, Maio 1999.
- [18] M.A. Maia e R.S. Bigonha. Unificando os Modelos de Computação Móvel e Componentes de Software via Máquinas de Estado Interativas. submitted to II Workshop on Formal Methods, Florianopolis, 1999.
- [19] Peter Wegner. Interactive software technology. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1997. To appear.
- [20] Roberto S. Bigonha, *The Object Oriented Language **Script** for Denotational Semantics*. Relatório Técnico LLP 005/99. Universidade Federal de Minas Gerais, Belo Horizonte, 1999. Available from <http://www.dcc.ufmg.br/bigonha/Llp/llp.html>.
- [21] Blakley, B. A Smalltalk Evolving Algebra and its Uses; PhD thesis, University of Michigan; Ann Arbor; Michigan; 1992.
- [22] Bvrger E; A Logical Operational Semantics for full Prolog, Part I; Selection Core and Control. In E. Bvrger, H. Kleine Büning, M. M. Richter, and W. Schvnfeld, editors; CSL`89. 3rd Workshop on Computer Science Logic, c olume 440 of LNCS; pages 36-64; Springer; 1990.
- [23] Bvrger E; A Logical Operational Semantics for full Prolog, Part II; Built-in Predicates for Database Manipulation; in B. Rován, editor; Mathematical Foundation of Computer Science; volume 452 of LNCS; pages 1-14; Springer; 1990.
- [24] Bvrger E; 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.

- [25] Gordon, Michael J. C.; The Denotational Description of Programming Languages - An Introduction; Springer-Verlag; New York; 1979.
- [26] Gurevich, Y.; Moss, L.; Algebraic Operational Semantics and Occam; in E. Bvrger, H. Kleine Büning, and M. M. Richter, editors; CSL`89, 3rd Workshop on Computer Science Logic; volume 440 of LNCS; pages 176-192; Springer; 1990.
- [27] Gurevich, Y.; Evolving Algebras:An Attempt to Discover Semantics; in Current Trends in Theoretical Computer Science, G. Rozenberg, A. Salomaa, eds., pages266-292; 1993.
- [28] Gurevich, Y.; Huggins, J. K.; The Semantics of the C Programming Language; Springer Lecture Notes in Computer Science 702, pages 274-308; 1993.
- [29] Gurevich, Y.; Evolving Algebras 1993: Lipari guide. In E. Bvrger, editor, Specification and Validation Methods, pages 9-63; Oxford University Press, 1995.
- [30] Di Iorio, V. O.; Bigonha, R. da S.; Maia, M. de A.; A Self-Applicable Partial Evaluator for ASM; In Proceedings of the Annual ASM User Group Meeting at the World Congress on Formal Methods for the Development of Computing System; Tolouse; França; September; 1999.
- [31] Jensen, K.; Wirth, N.; PASCAL User Manual and Report; Springer-Verlag; 1974.
- [32] Knuth, D. E.; The Remaining Trouble Spots in ALGOL 60; Communications of the ACM; Vol. 10, No. 10; 1967.
- [33] Maia, M de A.; Bigonha R. da S., Interaction-Based Semantics for Mobile Objects; Technical Report LLP 02/99; Laboratório de Linguagens de Programação; UFMG; 1999.
- [34] Maia, M. de A.; Di Iorio, V. O.; Bigonha, R. da S.; An ASM-based Approach for Connection-oriented Systems; In Proceedings of the Annual ASM User Group Meeting at the World Congress on Formal Methods for the Development of Computing System; Tolouse; França; September; 1999.
- [35] Morris, J.; Algebraic Operational Semantics and Modula-2; PhD thesis, University of Michigan; Ann Arbor; Michigan, 1988.
- [36] Nicholson, T.; Foo, N. A Denotational Semantics for Prolog; ACM Transactions on Programming Languages and Systems, Vl. 11, nro 4; pages 650-665; October; 1989.
- [37] Tirelo, F.; Maia, M. de A.; Di Iorio, V. O.; Bigonha, R da S.; Máquinas de Estado Abstratas; III SBLP; Porto Alegre; 1999.

- [38] Tirelo, F.; Maia, M. de A.; Di Iorio, V. O.; Bigonha, R da S.; *Machina: A Linguagem de Especificação de ASM*, Technical Report LLP 07/99; Laboratório de Linguagens de Programação; UFMG; 1999.
- [39] Wallace, C.; The Semantics of the C++ Programming Language; in E. Börger, editor, *Specification and Validation Methods*; pages 131-164; Oxford University Press; 1995.
- [40] Wallace, C.; The Semantics of the Java Programming Language: Preliminary Version; Technical Report CSE-TR-355-97, EECS Dept.; University of Michigan; December; 1997.
- [41] Welsh, J.; Sneeringer, W. J.; Hoare, C. A. R.; Ambiguities and Insecurities in PASCAL; *Software Practice and Experience*; 1977.
- [42] Börger, E. High Level System Design and Analysis using Abstract State Machines. In D. Hutter and Stephan and Traverso and Ullmann, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, LNCS. Springer-Verlag, 1999.
- [43] Börger, E. & Schulte, W. 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.
- [44] Del Castillo, G., Durdanovic, I. and Gldsser, U. *The Evolving Algebra Interpreter Version 2.0*. pages 191-214, 1996. Available from <http://www.eecs.umich.edu/gasm>.
- [45] Del Castillo, G. 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.
- [46] Gurevich, Y. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9-36. Oxford University Press, 1995.
- [47] Gurevich, Y. and Huggins, J. The Semantics of the C Programming Language, In E. Börger, H Kleine Büning, G. Jdger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of LNCS, pages 274-309. Springer, 1993.
- [48] Tirelo, F., Maia, M.A., Iorio, V.O., Bigonha, R.S. *Máquinas de Estado Abstratas*. Tutorial apresentado no III Simpósio Brasileiro de Linguagens de Programação. Porto Alegre, 1999. Available from <http://www.dcc.ufmg.br/bigonha/Llp/llp.html>
- [49] Tirelo, F., Bigonha, R.S., Maia M.A., Iorio, V.O., *Linguagem de Programação Algébrica Machina*, RT07/99, Laboratório de Linguagens de Programação, DCC/UFMG, 1999. Available from <http://www.dcc.ufmg.br/bigonha/Llp/llp.html>

- [50] Lucas Martins Amaral and Roberto S. Bigonha. Análise e projeto de ambientes interativos. Technical Report 02/96, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Janeiro 1996.
- [51] Lucas Martins Amaral and Roberto S. Bigonha. Xlds: Ambiente interativo para definição de semântica. Technical Report 03/96, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Janeiro 1996.
- [52] M.T.O Valente and Roberto S. Bigonha. A linguagem de programação ita. Technical Report 05/96, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Janeiro 1996.
- [53] G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra Abstract Machine. In H. K. 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.
- [54] A. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.
- [55] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
- [56] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [57] Y. Gurevich and J. Huggins. Evolving Algebras and Partial Evaluation. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 587–592, Elsevier, Amsterdam, the Netherlands, 1994.
- [58] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- [59] N. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.
- [60] M. Maia and R. Bigonha. The Formal Specification of the Interactive Abstract State Machines Languages. Technical Report 005/98, Departamento de Ciência da Computação, UFMG, 1998.
- [61] M. Maia, V. Iorio, and R. Bigonha. Interacting Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

- [62] Vladimir O. Di Ioro, Roberto S. Bigonha, and Marcelo A. Maia. A self-applicable partial evaluator for asm. In *Proceedings of the Annual ASM User Group Meeting at the World Congress on Formal Methods for the Development of Computing System*, pages 20–21, Toulouse, França, September 1999.
- [63] Marcelo A. Maia, Vladimir O. Di Ioro, and Roberto S. Bigonha. An asm-based approach for connection-oriented systems. In *Proceedings of the Annual ASM User Group Meeting at the World Congress on Formal Methods for the Development of Computing System*, pages 20–21, Toulouse, França, September 1999.
- [64] Marcelo A. Maia and Roberto S. Bigonha. Interaction based-semantics for mobile objects. In *Proceedings of the III Simpósio Brasileiro de Linguagens de Programação*, pages 93–107, Porto Alegre, RS, Maio 1999. Sociedade Brasileira de Computação.
- [65] Marcelo A. Maia and Roberto S. Bigonha. Rules for verification of interactive systems. In *Proceedings of the III Simpósio Brasileiro de Linguagens de Programação*, pages 207–212, Porto Alegre, RS, Maio 1999. Sociedade Brasileira de Computação.
- [66] O. Di Ioro, Vladimir and Roberto S. Bigonha. Avaliação parcial de máquinas de estado abstratas. In *Proceedings of the III Simpósio Brasileiro de Linguagens de Programação*, pages 207–212, Porto Alegre, RS, Maio 1999. Sociedade Brasileira de Computação.
- [67] Bigonha, Roberto S., *Literate Denotational Semantics*, Technical Report of the UFMG CS Department (in preparation), 1994.
- [68] Bigonha, Roberto S., **Script**: *An Object Oriented Language for Denotational Semantics*, Technical Report # 15/94, UFMG CS Department, 1994.
- [69] E. Börger e I. Durdanovic. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
- [70] G. Castillo, I. Durdanovic, e Glässer. An evolving algebra abstract machine. In H.K. Böning, editor, *Computer Science Logic, Selected papers from CSL’95*, páginas 191–214. Springer Lecture Notes in Computer Science 1092, 1996.
- [71] Y. Gurevich e J.K Huggins. The semantics of the C programming language. In *Springer Lecture Notes in Computer Science 702*, páginas 274–308, 1993. First appeared in *Selected papers from CSL’92 (Computer Science Logic)*.
- [72] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, páginas 9–36. Oxford University Press, 1995.
- [73] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, agosto 1978. Reprinted in “Distributed Computing: Concepts and Implementations” edited by McEntire, O’Reilly and Larson, IEEE, 1984.

- [74] J. Huggins. The Abstract State Machine Home Page at Michigan, <http://www.eecs.umich.edu/gasm>.
- [75] Naoki Kobayashi e Akinori Yonezawa. Towards foundations for concurrent object-oriented programming – types and language design –. *Theory and Practice of Object Systems*, John Wiley & Sons, 1995. to appear.
- [76] P.W. Kutter e A. Pierantonio. The formal specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.
- [77] P.W. Kutter e A. Pierantonio. Montages: Specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
- [78] M.A. Maia e R.S. Bigonha. Objetos Concorrentes em Máquina de Estados Abstratos Distribuída. In *II Simpósio Brasileiro de Linguagens de Programação*, Setembro 1997.
- [79] Robin Milner. *A Calculus of Communicating Systems*, volume 92 de *Lecture Notes in Computer Science*. Springer Verlag, 1980.
- [80] Michael Papathomas. A unifying framework for process calculus semantics of concurrent object-oriented languages. In M. Tokoro, O. Nierstrasz, e P. Wegner, editores, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, páginas 53–79. Springer-Verlag, 1992.
- [81] Benjamin C. Pierce e David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito e Akinori Yonezawa, editores, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, número 907 in *Lecture Notes in Computer Science*, páginas 187–215. Springer-Verlag, abril 1995.
- [82] Benjamin C. Pierce. Foundational calculi for programming languages. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*, capítulo 139. CRC Press, 1996. To appear.
- [83] C. Priami. *Enhanced Operational Semantics for Concurrency*. Tese de Doutorado, Università di Pisa, Pisa, march 1996.
- [84] C. Wallace. The semantics of the C++ programming language. In E. Bvrger, editor, *Specification and Validation Methods*, páginas 131–164. Oxford University Press, 1995.
- [85] M. J. C. Gordon, *The Denotational Description of Programming Languages – An Introduction*, Springer–Verlag, 1976.
- [86] C. A. Gunter, P. D. Mosses and D. S. Scott, *Semantic Domains and Denotational Semantics*, Internal Report DAIMI PB –276, Computer Science Department, Aarhus University, April 1989.

- [87] P. Mosses, *SIS – Semantics Implementation System, Reference Manual and User Guide*, Technical Report DAIMI MD-30, Aarhus University, August 1979
- [88] P. Mosses, *Mathematical Semantics and Compiler Generations*, Ph. D Thesis, Oxford University, 1975.
- [89] D. A. Schmidt, *Denotational Semantics : A Methodology for Language Development*, Allyn & Bacon, 1986.
- [90] J. E. Stoy, *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, 1977.
- [91] D. S. Scott, *Toward a Mathematical Semantics for Computer Languages*, Technical Monograph, PRG-6, Oxford University, Computing Laboratory, 1971.
- [92] D. S. Scott, *Outline of a Mathematical Theory of Computation*, Technical Monograph, PRG-2, Oxford University, Computing Laboratory, 1971.
- [93] R. D. Tennent, “The Denotational Semantics of Programming Languages”, Communication of the ACM, August 1976, pp 437 – 453.
- [94] M. Tofte, *Compiler Generators*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1990
- [95] Augustsson, L. *A compiler for lazy ML*. Em Proc. of the ACM Symposium on Lisp and Functional Programming. Austin., pp. 218-27. Agosto.
- [96] Burns, G.L., Peyton Jones, S.L., Robson, J.D. *The spineless g-machine*. Em Proc. ACM Conference on Lisp and Functional Programming. pp.244-58. Snowbird. USA.
- [97] Cardelli, L. *Compiling a functional language*. Em Proceedings of the ACM Symposium on Lisp and Functional Programming. Austin. pp.208-17. Agosto.
- [98] Fairbairn, J., Wray, S. *TIM: A simple, lazy abstract machine to execute supercombinators*. Em Proc. of Third International Conference on Functional Programming and Computer Architecture. pp.34-45. LNCS 274. Springer Verlag.
- [Henderson, 1980] Henderson, P. *Functional Programming: application and implementation*. Prentice Hall.
- [99] Johnsson, T. *Efficient compilation of lazy evaluation*. Em Proc. of the ACM Conference on Compiler Construction. Montreal. pp.58-69. Junho.
- [100] Landin, P.J. *The mechanical evaluation of expressions*. Computer Journal. Vol. 6. pp.308-20.

- [101] Lins, R.D. *Categorical multi-combinators*. Em Gilles Kahn (ed.). Functional Programming Languages and Computer Architecture. Springer Verlag, LNCS 274. pp.60-79.
- [102] Lins, R.D., Lira, B.O. Γ CMC: *A Novel Way of Compiling Functional Languages*. Em Anais do XII Congresso da Sociedade Brasileira de Computação.
- [103] Mosses, P.D. *Mathematical Semantics and Compiler Generation*. PhD thesis, Oxford University Computing Lab. Programming Research Group.
- [104] Mosses, P.D. *SIS - A compiler-generator system using denotational semantics*. Technical report. University of Aarhus. Denmark.
- [105] Musicante, M.A., Lins, R.D. *GMC: A Graph Multi-Combinator Machine*. Microprocessing and Microprogramming. pp.31-35. Abril.
- [106] Peyton Jones, S.L. *The implementation of functional languages*. Prentice Hall.
- [107] Peyton Jones, S.L., Salkild, J. *The spineless tagless g-machine*. Em Proc. ACM Conference on Functional Programming Languages and Computer Architecture. pp.184-201. Snowbird. USA.
- [108] Turner, D. *A new implementation technique for applicative languages*. Software - practice and experience. Vol. 9. pp.31-49.
- [109] Bigonha, Roberto S., *Semântica Denotacional Legível*, VII Simpósio Brasileiro de Engenharia de Software, Rio de Janeiro, Brazil, 1993.