V SBLP - Submissão de Artigo Técnico

Extensões de Geração para Máquinas de Estado Abstratas

Vladimir O. Di Iorio
Departamento de Informática, Universidade Federal de Viçosa,
email: vladimir@dcc.ufmg.br

Roberto S. Bigonha
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais,
email: bigonha@dcc.ufmg.br

Resumo

As *Máquinas de Estado Abstratas* (ASM, do inglês *Abstract State Machine*s) são um formalismo criado com o objetivo de simular algoritmos em uma maneira direta, no nível de abstração desejado. Uma *extensão de geração* para um programa P é um programa que, recebendo parte dos dados de entrada de P, produz um novo programa designado por *residual* ou *especializado*. O programa residual, se executado sobre o restante da entrada, gera os mesmos resultados que o programa original, se executado sobre a entrada completa. Neste artigo, divulgamos técnicas usadas para desenvolver um *gerador de extensões de geração* para a linguagem ASM. A aplicação do gerador de extensões de geração à descrição da semântica de uma linguagem L, escrita em ASM, produz automaticamente um compilador de L para ASM. Experimentos envolvendo geração de compiladores são apresentados.
Palavras-Chave: Máquinas de Estado Abstratas, avaliação parcial, extensões de geração.

Abstract

The *Abstract State Machines* (ASM, for short) are a formalism created to model algorithms at its natural abstraction level. A *generating extension* of a program P is a program which, when given part of P's input data, produces a *residual* or *specialized* program. The residual program, when given the remaining input data, yields the same result that the original program produces, when given all input data. In this work, we present techniques to develop a *generating extension generator* for the ASM language. A compiler from a language L to ASM can be automatically generated by submiting a description of the semantics of L, written in ASM, to the generating extension generator. Experiments involving compiler generation are described.
Key words: Abstract State Machines, partial evaluation, generating extensions.

Extensões de Geração para Máquinas de Estado Abstratas

Resumo *Máquinas de Estado Abstratas* (ASM, do inglês *Abstract State Machines*) são um formalismo criado com o objetivo de simular algoritmos em uma maneira direta, no nível de abstração desejado. Uma *extensão de geração* para um programa P é um programa que, recebendo parte dos dados de entrada de P, produz um novo programa designado por *residual* ou *especializado*. O programa residual, se executado sobre o restante da entrada, gera os mesmos resultados que o programa original, se executado sobre a entrada completa. Neste artigo, divulgamos técnicas usadas para desenvolver um *gerador de extensões de geração* para a linguagem ASM. A aplicação do gerador de extensões de geração à descrição da semântica de uma linguagem L, escrita em ASM, produz automaticamente um compilador de L para ASM. Experimentos envolvendo geração de compiladores são apresentados.
Palavras-Chave: Máquinas de Estado Abstratas, avaliação parcial, extensões de geração.

1 Introdução

Máquinas de Estado Abstratas (*ASM*, do inglês *Abstract State Machines*) são um formalismo criado por Yuri Gurevich [10], utilizado na descrição da semântica de linguagens de programação [5, 11], arquiteturas de sistemas [4], protocolos distribuídos [13], 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.

Seja P um programa com entradas in_1 e in_2. Um *avaliador parcial* é um programa que, quando executado sobre P e a entrada in_1, produz um novo programa P_{in_1}, chamado de *programa residual* ou *especializado* [15]. O programa residual recebe apenas uma entrada in_2 e produz os mesmos resultados que P, se lhe fossem fornecidas ambas as entradas. As entradas in_1 e in_2 são designadas, respectivamente, *estática* e *dinâmica*.

Uma alternativa para avaliação parcial é conhecida como *abordagem de extensões de geração*. Uma *extensão de geração* para o programa P descrito acima é um programa P_{gen} que, quando executado sobre in_1, produz um programa residual P_{in_1}. Um *gerador de extensões de geração* é um programa que constrói extensões de geração. As duas abordagens permitem obter resultados equivalentes, mas existem razões para se utilizar uma ou outra, que discutiremos na Seção 3.

Neste trabalho apresentamos um gerador de extensões de geração para a linguagem das Máquinas de Estado Abstratas. O gerador de extensões de geração possui um núcleo com rotinas que podem ser aplicadas a qualquer linguagem que utilize o modelo ASM, se for construída uma interface apropriada. Como linguagem concreta, utilizamos nos testes uma versão de ASM conhecida como Xasm, que será introduzida junto com o modelo ASM, na Seção 2.

Como vimos acima, a descrição da semântica de uma linguagem L pode ser dada por um interpretador Int escrito em ASM. Quando o gerador de extensões de geração é aplicado a Int, produz-se uma extensão de geração que é na realidade um compilador de L para ASM. Na Seção 4, descrevemos as técnicas utilizadas para a construção do gerador de extensões de geração para ASM. Na Seção 5, experimentos envolvendo geração de compiladores são discutidos.

O objetivo principal do trabalho desenvolvido é o ganho em eficiência obtido com a especialização. A maioria dos modelos para descrição de semântica possui implementações pouco eficientes. Com a geração de programas especializados, procuramos tornar mais atraente a utilização do modelo ASM.

2 ASM Seqüenciais

As Máquinas de Estado Abstratas constituem um formalismo poderoso o bastante para especificar de maneira simples algoritmos envolvendo processamento paralelo e distribuído. Entretanto, nosso principal objetivo envolve sua utilização na especificação da semântica de linguagens de programação seqüenciais.

Esta seção descreve o subconjunto das ASM designado por *ASM Seqüenciais*. As definições visam tornar o artigo autocontido, mas uma abordagem mais completa e formal sobre ASM pode ser encontrada em [10] e [6].

2.1 Definições

A *assinatura* de uma ASM seqüencial $\mathcal{A}$ 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 em funções sobre os elementos do superuniverso. As interpretações são designadas *funções básica*s 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 será 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, $f(\bar{a}) =$*undef* significa que f é indefinida para a tupla $\bar{a}$. Uma relação *r-ária* sobre X pode ser vista como uma função $X^r \rightarrow \{$*true*,*false*$\}$. Um *universo* U é um tipo especial de função básica: uma relação unária geralmente identificada pelo conjunto dos elementos x tais que $U(x) =$*true*, i.e., $\{x : U(x)\}$.

Em princípio, um programa de $\mathcal{A}$ é uma regra de transição, que pode ser *regra de atualização*, *construtor de bloco* ou *construtor condicional*.

Uma regra de atualização tem o formato $f(\overline{t})$ `:=` t_0, onde f é o nome de uma função da assinatura de $\mathcal{A}$, $\overline{t}$ 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 é a seguinte: a tupla $\overline{t}$ 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` g_0 `then` R_0 `elseif` g_1 `then` R_1 ... `elseif` g_k `then` R_k `endif`

Sua semântica é a seguinte: uma regra R_i, $0 \leq i \leq k$, é executada se os termos booleanos $g_0, ..., g_{i-1}$ 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 determinística 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 determinam o final da execução quando o disparo da regra de transição não produz nenhuma atualização, ou então um comando especial indica explicitamente o término da execução.

Para permitir uma interface com o mundo externo, o modelo ASM oferece *funções externas*. Uma função externa não precisa retornar necessariamente um mesmo valor para chamadas com os mesmos parâmetros, se essas forem disparadas em passos diferentes de uma execução. Outras regras de transição podem ser utilizadas em um programa ASM, algumas utilizando variáveis. Por exemplo, a regra `choose` permite escolha não determinística de um elemento de um conjunto, e a regra `var` permite execução em paralelo de uma outra regra, instanciando uma variável com elementos de um dado universo.

2.2 A Linguagem Xasm

A linguagem Xasm foi criada por Matthias Anlauff [2] e se baseia no modelo ASM. Além de regras básicas ASM, oferece uma série de extensões para a linguagem das Máquinas de Estado Abstratas. Possui um compilador para a linguagem C, o que permite executar as especificações sem a necessidade de um interpretador.

Uma característica interessante de Xasm é a possibilidade de criar abstrações de funções. Essas funções podem ser executadas em vários passos, independente da especificação em que estão inseridas. A linguagem permite a execução de comandos seqüencialmente, por meio de um construtor especial.

Outra característica interessante de Xasm é a possibilidade de especificar uma gramática que descreve uma linguagem L qualquer. Um analisador léxico e um analisador sintático são automaticamente gerados. Um programa escrito em L pode ser lido e convertido para um formato interno, para então ser processado. Isso facilita a utilização em descrições de semântica de linguagens de programação, uma das mais importantes aplicações de ASM.

2.3 Um Interpretador escrito em Xasm

A Figura 1 exibe um interpretador para uma versão da Máquina de Turing, escrito em Xasm. A máquina possui as instruções `right`, `left`, `write a`, `goto i`, `if a goto i`, `stop`. Uma fita infinita, representada pela função `tape`, armazena elementos do conjunto $\{0, 1, \mathit{undef}\}$. A instrução corrente do programa MT é indicada por `pc` e a cabeça de leitura/gravação é indicada por `thead`.

A primeira linha da especificação exibe o nome da mesma e os parâmetros. Neste caso, os parâmetros são nomes de arquivos que armazenam um programa MT (`progfile`) e o estado inicial da fita (`tapefile`). Uma gramática descreve o formato dos programas MT e gera automaticamente um analisador léxico e sintático, que é disparado pela função `TMparse`. As ações associadas à gramática definem os valores das funções `prog_instr`, `prog_par1` e `prog_par2`. Dado o número da instrução corrente `pc`, essas funções permitem acessar o código da instrução MT e seus parâmetros. A especificação é dividida em duas seções: inicialização e regra de transição. A seção de inicialização define os valores iniciais das funções, e a regra de transição é disparada a cada passo da execução.

3 Avaliação Parcial e Extensões de Geração

Na avaliação parcial tradicional, um programa e parte de sua entrada são submetidos a um avaliador parcial, que produz um programa residual. Dois métodos diferentes podem ser utilizados: avaliação parcial *online* e *offline*. Esses métodos diferem com relação ao momento em que os componentes do programa são classificados como estáticos e dinâmicos:

```
asm TURING (progfile, tapefile)

//... declaração de funções, especificação da gramática ...

init
  thead := READTAPE (tapefile), RootNode := TMparse (progfile),  pc := 0
endinit

if prog_instr (pc) = LEFT then
   pc := pc + 1,  thead := thead - 1
elseif prog_instr (pc) = RIGHT then
   pc := pc + 1,  thead := thead + 1
elseif prog_instr (pc) = WRITE then
   pc := pc + 1,  tape (thead) := prog_par1 (pc)
elseif prog_instr (pc) = GOTO then
   pc := prog_par2 (pc)
elseif prog_instr (pc) = IFGOTO then
   if tape (thead) = prog_par1 (pc) then  pc := prog_par2 (pc)
   else  pc := pc + 1  endif
elseif prog_instr (pc) = STOP then
   exit := 1
endif

endasm
```

Figura1. Interpretador para MT escrito em Xasm.

junto com a execução da especialização, na abordagem *online*, e em uma fase anterior, designada BTA (análise de tempo de definição), na abordagem *offline* [15].

Uma técnica diferente se tornou popular na década de 90, designada *abordagem de extensões de geração* [18, 1, 3]. Essa técnica consiste em escrever à mão um *gerador de extensões de geração*, que a princípio poderia ser produzido automaticamente pela auto-aplicação de um avaliador parcial tradicional. Um gerador de extensões de geração, geralmente chamado de `cogen`, quando aplicado a um programa P e informações que indiquem que parte da entrada de P é estática, produz uma *extensão de geração* para P.

A Figura 2 exibe um esquema de uso de um avaliador parcial tradicional e de `cogen`. Retângulos com bordas retas representam programas, e com bordas arrendondadas representam dados. Setas simples indicam entrada de dados, enquanto que setas duplas indicam saída produzida por um programa.

3.1 Definições Equacionais

Se P é um programa escrito em uma linguagem S, vamos usar $[\![P]\!]_S$ para denotar a semântica de P. As equações exibidas na Figura 3(a) descrevem o funcionamento de um avaliador parcial `mix` para programas da linguagem S. O programa P é escrito em uma linguagem S e, quando aplicado às entradas in_1 e in_2, gera a saída *out*. Um avaliador parcial `mix`, quando aplicado a P e parte de sua entrada, gera um programa residual P_{in_1}. O programa residual, se executado sobre o restante da entrada, gera a mesma saída que o programa original, se executado sobre ambas as entradas. O avaliador parcial `mix` é escrito em uma linguagem L e o programa residual é gerado em uma linguagem T. Geralmente, S e T são a mesma linguagem.

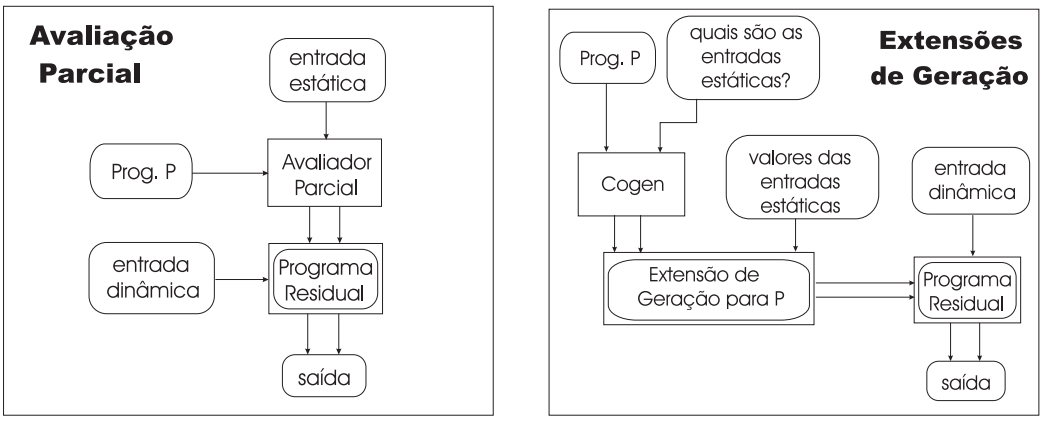

Figura2. Esquema de Uso de um Avaliador Parcial e de `Cogen`.

(a)	$out = [\![P]\!]_S (in_1, in_2)$ $P_{in_1} = [\![\texttt{mix}]\!]_L (P, in_1)$ $out = [\![P_{in_1}]\!]_T (in_2)$	(b)	$P_{gen} = [\![\texttt{cogen}]\!]_L (P, \{in_1 \approx STA,\ in_2 \approx DYN\})$ $P_{in_1} = [\![P_{gen}]\!]_S (in_1)$
(c)	Primeira Projeção: $[\![\texttt{mix}]\!]_L (P, in_1) = P_{in_1}$ Segunda Projeção: $[\![\texttt{mix}]\!]_L (\texttt{mix}, P) = P_{gen}$ Terceira Projeção: $[\![\texttt{mix}]\!]_L (\texttt{mix},\texttt{mix}) = \texttt{cogen}$		

Figura3. Definições Equacionais.

Um gerador de extensões de geração `cogen` tem como entradas um programa P e informações que indiquem quais entradas de P são estáticas (STA) e dinâmicas (DYN). Produz uma extensão de geração P_{gen} para P, como exibido na Figura 3(b).

3.2 Equivalência das Abordagens

As abordagens de avaliação parcial tradicional e extensões de geração são, sob um certo ponto de vista, equivalentes. Futamura foi o primeiro a observar que o programa `cogen` pode ser automaticamente gerado por meio da auto-aplicação de um avaliador parcial `mix` [9]. As equações que incluem esse resultado são conhecidas como as Três Projeções de Futamura, exibidas na Figura 3(c). Se P for um interpretador de uma linguagem L' e in_1 for um programa escrito nessa linguagem, então P_{in_1} será o resultado da compilação de in_1. A linguagem objeto será T, a linguagem dos programas gerados pelo avaliador parcial. O programa P_{gen} será um compilador de $Lint$ para T e `cogen` pode ser usado como um gerador de compiladores. Por isso usamos o nome `cogen` para designar um gerador de extensões de geração.

A Terceira Projeção de Futamura mostra que um gerador de extensões de geração `cogen` pode ser produzido por meio da auto-aplicação de um avaliador parcial. Por outro lado, `cogen` pode ser usado para obter os mesmos resultados da avaliação parcial tradicional, com um passo adicional. Produzindo uma extensão de geração para P e aplicando-a à entrada estática de P, tem-se um programa especializado, equivalente ao obtido com um avaliador parcial tradicional.

Assim, as abordagens podem ser consideradas equivalentes. Então qual delas deve ser adotada? Se não formos considerar auto-aplicação, construir um avaliador parcial tradicional é uma tarefa mais simples do que construir um gerador de extensões de geração.

Entretanto, aplicações como geração de compiladores usando avaliação parcial tradicional requerem auto-aplicação. Algumas razões podem justificar então a adoção da abordagem de extensões de geração:

1. O gerador de extensões de geração pode ser escrito em outra linguagem, de nível mais alto, do que a linguagem dos programas que ele processa. Por outro lado, um avaliador parcial auto-aplicável deve ter o poder de processar o seu próprio texto. Isso pode trazer dificuldades adicionais na construção do avaliador parcial.
2. Um avaliador parcial deve conter um meta-interpretador, o que pode ser um problema para linguagens fortemente tipadas. Nem o gerador de extensões de geração, nem as extensões de geração produzidas, precisam conter um meta-interpretador.

3.3 ASM e Auto-Aplicação de `mix`

Em [12], Huggins e Gurevich apresentam um avaliador parcial para ASM, escrito na linguagem C. Esse avaliador parcial é muito simples e não permite a auto-aplicação, assim experimentos com geração de compiladores não são conduzidos.

Em [7], apresentamos um avaliador parcial para ASM mais poderoso, escrito em Java, e uma segunda versão, escrita na própria linguagem ASM, que chamamos de $\texttt{mix}_{ASM}$. O funcionamento detalhado de $\texttt{mix}_{ASM}$ é discutido em [8], e experimentos com geração de compiladores simples são conduzidos usando a Segunda Projeção de Futamura:

$$[\![\texttt{mix}]\!]_{\text{Java}}\,(\texttt{mix}_{ASM}, \mathit{Interpreter}) = \mathit{Compiler}$$

O avaliador parcial escrito em Java é usado para especializar a versão escrita na linguagem ASM. Para conseguirmos bons resultados na especialização, fomos obrigados a utilizar na implementação de $\texttt{mix}_{ASM}$ uma regra ASM não básica, conhecida como regra *var*. Sua semântica é o disparo em paralelo de um conjunto de regras ASM, instanciando variáveis com cada elemento de um dado domínio. O avaliador parcial escrito em Java processa essas construções corretamente, mas não fomos capazes de expressar esse processamento no próprio $\texttt{mix}_{ASM}$ sem afetar negativamente os resultados da especialização. Assim, o poder dos dois avaliadores parciais não é o mesmo, não configurando exatamente uma auto-aplicação.

A conseqüência negativa dos resultados apresentados acima é que não foi possível então aplicar a Terceira Projeção de Futamura. Como $\texttt{mix}_{ASM}$ deve servir de entrada a si próprio, teria que processar todas as construções utilizadas na sua implementação. Isso foi a principal motivação para adotarmos então a abordagem de extensões de geração. Como vimos na Seção 3.2, o gerador de extensões de geração `cogen` pode ser escrito em um linguagem de nível mais alto do que a dos programas que ele processa, evitando os problemas associados à auto-aplicação.

4 Cogen para ASM

Nesta seção, vamos apresentar um gerador de extensões de geração `cogen` para a linguagem das Máquinas de Estado Abstratas. O programa `cogen` para ASM é composto de duas partes bem separadas:

***Núcleo*:** recebe como entradas uma representação abstrata de uma especificação ASM $\mathcal{S}$ e uma indicação de quais entradas de $\mathcal{S}$ são estáticas. Produz como saída uma representação abstrata de uma extensão de geração para a especificação de entrada $\mathcal{S}$.

$R ::= \quad f(t_1, \ldots, t_r) := t$	$term : (FNAME \times TERM^*) \rightarrow TERM$
$R ::= \quad R_1 \ldots R_k$	$update : (TERM \times TERM) \rightarrow RULE$
$R ::= \quad$ `if` t `then` R_1 `else` R_2 `endif`	$block : RULE^* \rightarrow RULE$
$t ::= \quad f(t_1, \ldots, t_r)$	$cond : (TERM \times RULE \times RULE) \rightarrow RULE$
	$condsta : (TERM \times GERULE \times GERULE) \rightarrow GERULE$
	$dsnode : RULE^* \times RULE \rightarrow GERULE$
	$gotostate : INT \rightarrow RULE$

Figura4. Sintaxe Abstrata de Especificações e Representação Interna.

***Filtros*:** são conjuntos de rotinas que permitem utilizar o núcleo de `cogen` com uma linguagem ASM concreta específica. Contêm rotinas para leitura de uma especificação a partir de um arquivo, conversão para o formato utilizado pelo núcleo e geração de uma extensão de geração na linguagem concreta a partir da representação abstrata construída pelo núcleo. Nas experiências conduzidas, utilizamos um filtro para a linguagem Xasm.

Nesta seção, os algoritmos que implementam o núcleo de `cogen` são descritos usando o próprio modelo ASM. Mas isso não sugere qualquer intenção de auto-aplicação, o único objetivo é deixar claro o funcionamento dos algoritmos. Os algoritmos processam especificações ASM, assim precisamos estabelecer uma notação para a representação da especificação de entrada e da extensão de geração que será produzida.

4.1 Representação de Especificações

O gerador de extensões de geração processa apenas regras de transição ASM básicas, isto é, instruções de atualização, construtores condicionais e blocos de regras. A sintaxe abstrata dessas regras é apresentada no lado esquerdo da Figura 4, onde f denota nomes de funções, t denota termos e R denota regras.

Para representar as especificações ASM que servem como entrada para `cogen`, vamos utilizar definições similares às apresentadas em [6]. Os componentes sintáticos são elementos dos domínios *FNAME* (nomes de funções), *TERM* (termos) e *RULE* (regras de transição). São representados usando-se os construtores *term*, *update*, *block* e *cond*, como descrito no lado direito da Figura 4. Para a representação abstrata da extensão de geração, que é a saída produzida por `cogen`, um construtor *gotostate* será também utilizado junto com o domínio *RULE*. Além disso, será necessário um novo domínio, designado *GERULE*, e os construtores *condsta* e *dsnode*. Na Seção 4.3, veremos exatamente como esses construtores são utilizados.

A extensão de geração é uma especificação que gera como saída outra especificação ASM. Assim, dentro de seu código existem chamadas para funções externas que constroem regras ASM. Independente da linguagem concreta utilizada, qualquer extensão de geração irá utilizar o mesmo conjunto de funções externas para a construção da especificação residual. O núcleo oferece a implementação de um procedimento de compressão de transições automático, otimizando o programa residual. O filtro para uma determinada linguagem ASM concreta deve apenas implementar uma tradução direta da representação abstrata otimizada para a linguagem concreta em questão.

```
preproc : RULE → RULE

preproc (update (loc,val)) ≡  update (loc,val)
preproc (cond (c, r1, r2)) ≡  cond (c, preproc(r1), preproc(r2))
preproc (block (⟨r1, ... , rk⟩)) ≡
  if rj = cond (c, s1, s2), para algum j in 1, ... , k then
    let s3 = merge_blocks (s1, block (⟨r1, ... , rj−1, rj+1, ... , rk⟩))
        s4 = merge_blocks (s2, block (⟨r1, ... , rj−1, rj+1, ... , rk⟩))
    in cond (c, preproc(s3), preproc(s4))  endlet
  else  block (⟨r1, ... , rk⟩)  endif
```

Figura5. Pré-processamento.

```
if (∃x ∈ BSET) then
  choose x ∈ BSET
    BSET(x):= false, ProcessBTA(x)
  endchoose
endif
```

```
if ¬(∃x ∈ BSET) and change then
  BSET(init):= true
  BSET(prog):= true
  change:= false
endif
```

Figura6. Regras para computar uma divisão BTA.

4.2 Pré-Processamento e Análise de Tempo de Definição

Para simplificar a codificação da extensão de geração, a regra de transição das especificações ASM será pré-processada. A Figura 5 exibe uma definição formal para esse procedimento, usando o paradigma funcional de avaliação estrita. A função `preproc(`r`)` recebe a regra de transição e retorna uma nova regra, com um formato de uma árvore, onde os nodos internos indicam regras condicionais e as folhas são blocos de atualizações. A função `merge_blocks`(r_1, r_2) constrói um bloco contendo as regras de r_1 e r_2.

Após o pré-processamento, a fase de análise de tempo de definição (BTA) é conduzida. Inicialmente, o usuário deve atribuir valores BTA (*STA* e *DYN*) às funções de entrada, indicando quais entradas são estáticas e quais são dinâmicas. O objetivo desta fase é determinar uma *divisão BTA*, ou seja, classificar todas as funções não-externas como estáticas ou dinâmicas, dependendo da sua relação com as funções de entrada.

A Figura 6 exibe as regras que formalizam a computação de uma divisão BTA. O método é conhecido como *interpretação abstrata*, pois funciona como uma execução da especificação, mas usando os valores *STA* (estático) e *DYN* (dinâmico) no lugar dos reais valores das funções.

O algoritmo de divisão é executado até que um ponto fixo seja alcançado. A função `init` obtém a regra de inicialização da especificação de entrada, enquanto que `prog` obtém a regra de transição. Uma relação unária `BSET` : $(RULE{+}TERM) \rightarrow BOOL$, de forma similar à adotada em [6], identifica quais instâncias de subregras ou termos estão sendo consideradas em um dado passo da execução. O valor inicial de `BSET` é $\{$`init`, `prog`$\}$. A função booleana `change` é utilizada para determinar quando um ponto fixo é alcançado. Seu valor inicial é *false*.

A cada passo, uma subregra ou termo é analisado pela "macro" `ProcessBTA`, cuja definição é exibida na Figura 7. Todos os termos $f(t_1, \ldots, t_r)$ da especificação são analisados. Se qualquer $t_i, 1 \leq i \leq r$ é dinâmico, então a função f deve ser classificada como dinâmica. Além disso, nas atualizações $f(t_1, \ldots, t_r) := t$, se t é dinâmico, então f

```
        if x = block (⟨r_1,...,r_k⟩) then
          var j ranges over 1..k
            BSET(r_j) := true
          endvar
        endif
```

```
        if x = cond (c, r_1, r_2) then
          BSET(c) := true
          BSET(r_1) := true
          BSET(r_2) := true
        endif
```

```
if x = update (term(f,t*), term(g,u*)) then
  BSET(term(f,t*)) := true
  BSET(term(g,u*)) := true
  if bta_val(g)=DYN & bta_val(f)=STA then
    bta_val(f) := DYN , change := true
  endif
endif
```

```
if x = term (f, ⟨g_1(t*_1),...,g_k(t*_k)⟩) & k > 0 then
var j ranges over 1..k
  BSET(g_j(t*_j)) := true
  if bta_val(g_j)=DYN & bta_val(f)=STA then
    bta_val(f) := DYN , change := true
  endif
endvar  endif
```

Figura7. Macro `ProcessBTA(`x`)`.

```
downdyn : RULE → RULE

downdyn (block (⟨r_1,...,r_k⟩)) ≡ block (⟨r_1,...,r_k⟩)
downdyn (update (t_1,t_2)) ≡ update (t_1,t_2)
downdyn (cond (term(f,t*), r_1, r_2)) ≡
  if bta_val(f) = DYN then ddcond (term(f,t*), downdyn(r_1), downdyn(r_2))
  else cond (term(f,t*), downdyn(r_1), downdyn(r_2))  endif

ddcond (c, block (r*_1), block (r*_2))) ≡ cond (c, block (r*_1), block (r*_2))
ddcond (c, cond (term(f,t*), r_1, r_2), r_3) ≡
  if bta_val(f) = STA then
    cond (term(f,t*), downdyn(cond(c,r_1,r_3)), downdyn(cond(c,r_2,r_3)))
  else cond (c, cond (term(f,t*), r_1, r_2), r_3)  endif
ddcond (c, r_3, cond (term(f,t*), r_1, r_2)) ≡
  if bta_val(f) = STA then
    cond (term(f,t*), downdyn(cond(c,r_3,r_1), downdyn(cond (c,r_3,r_2)))
  else cond (c, r_3, cond (term(f,t*), r_1, r_2))  endif
```

Figura8. Função `downdyn`.

deve também ser classificada como dinâmica. A função `bta_val` associa, a cada nome de função f da especificação de entrada, um valor *BTA* (*STA* ou *DYN*). Todas as funções são inicializadas com *STA*, exceto as funções de entrada e externas. O resultado final da divisão é dado pelos valores da função `bta_val`.

Após o final da BTA, a regra de transição de $\mathcal{S}$ é novamente processada. As regras condicionais dinâmicas podem sofrer uma troca de posição com as regras condicionais estáticas. A função `downdyn`, exibida na Figura 8, descreve esse processamento.

4.3 Representação Abstrata da Extensão de Geração

A especificação modificada pelo processamento discutido na Seção 4.2 é utilizada pelo núcleo de `cogen` para produzir a representação abstrata da extensão de geração. O formato da representação abstrata utiliza os construtores introduzidos na Seção 4.1.

A regra de transição da extensão de geração terá o formato de uma árvore. Caminhando da raiz para as folhas, encontra-se primeiro uma seqüência de testes relativos às regras condicionais classificadas como estáticas pela BTA. Esses testes são copiados diretamente para a extensão de geração, assim não farão parte da especificação residual. São representados por instâncias do construtor *condsta*. Observe, na Figura 4, que elementos

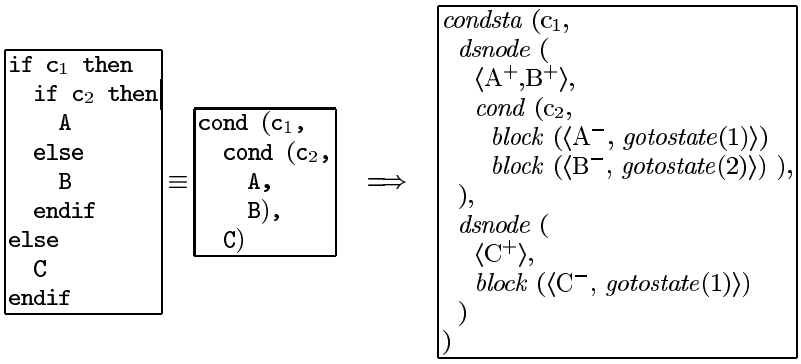

Figura9. Representação Abstrata em uma Extensão de Geração.

condsta podem ser aninhados recursivamente, até que um elemento representado por um construtor *dsnode* seja encontrado.

O construtor *dsnode* indica um código associado a um determinado fluxo de controle das regras condicionais estáticas. Cada instância representa uma diferente combinação da avaliação dessas condições estáticas, e está associada a uma subárvore que contém apenas regras condicionais dinâmicas e blocos de atualizações. Os elementos *dsnode* consistem de duas partes:

1. A primeira parte é uma lista de blocos. Cada bloco contém um conjunto de atualizações **estáticas** associado a cada diferente combinação da avaliação das condições **dinâmicas** presentes na subárvore.
2. A segunda parte representa a própria regra associada à subárvore, mas com todas as atualizações estáticas eliminadas. Essas atualizações foram coletadas e reunidas na primeira parte do elemento *dsnode*. No lugar de cada conjunto de atualizações estáticas, é inserido um construtor *gotostate* que indica extamente a qual conjunto está associado.

Para exemplificar esse processo, suponha que, no trecho de código exibido no lado esquerdo da Figura 9, a condição $\mathtt{c}_1$ seja estática e a condição $\mathtt{c}_2$ seja dinâmica. Suponha também que `A`, `B` e `C` sejam blocos de regras, contendo qualquer número de atualizações estáticas e dinâmicas. A representação usando construtores é exibida à direita do trecho de código. A estrutura mais à direita indica a representação abstrata associada ao código da esquerda, na extensão de geração produzida. É o resultado do processamento descrito nesta seção. Para indicar separadamente as atualizações estáticas e dinâmicas de um bloco `X`, a notação utilizada é $\mathtt{X}^+$ e $\mathtt{X}^-$, respectivamente.

A semântica é explicada a seguir. A extensão de geração, de forma semelhante a um avaliador parcial, mantém uma tabela com todos os possíveis estados gerados a partir de diferentes valores atribuídos às funções estáticas. No trecho acima, a condição estática $\mathtt{c}_1$ vai ser avaliada, decidindo qual dos ramos será executado. Se $\mathtt{c}_1$ for verdadeira, por exemplo, dois possíveis novos estados vão ser gerados, produzidos pelo disparo das atualizações estáticas de $\mathtt{A}^+$ e $\mathtt{B}^+$. O código residual gerado é uma modificação da regra condicional que contém $\mathtt{c}_2$. Observe que elementos *gotostate* são inseridos, com um valor inteiro que indica a posição a que estão associados na lista $\langle \mathtt{A}^+,\mathtt{B}^+ \rangle$.

```
sds : INT × RULE → GERULE

sds (n, block (⟨ ⟩)) ≡ dsnode (⟨block (⟨ ⟩)⟩, block (⟨gotostate(n)⟩))
sds (n, block (⟨update(term(f,t),t2), r2,... ,rk⟩)) ≡
  let dsnode (⟨b1,... ,bi⟩, block(⟨d1,... ,dj⟩)) = sds (n, block (⟨r2,... ,rk⟩) in
    if bta_val(f) = STA then
      dsnode (⟨merge(update(term(f,t),t2),b1), b2,... ,bi⟩, block(⟨d1,... ,dj⟩))
    else dsnode (⟨b1,... ,bi⟩, block(⟨update(term(f,t),t2), d1,... ,dj⟩))  endif
  endlet

sds (n, cond (term(f,t), r1, r2))) ≡
  if bta_val(f) = STA then condsta (term(f,t), sds(1,r1), sds(1,r2))
  else let dsnode (⟨b1,... ,bi⟩, r3) = sds(n,r1)
           dsnode (⟨c1,... ,cj⟩, r4) = sds(n+i,r2) in
    dsnode (⟨b1,... ,bi,c1,... ,cj⟩, cond (term(f,t), r3, r4))
  endlet  endif
```

Figura10. Função sds.

O algoritmo que descreve o processamento discutido nesta seção é definido pela função sds, exibida na Figura 10. A chamada inicial deve ser $\texttt{sds}(1, r)$, onde r é a regra de transição pré-processada.

4.4 Uso de uma Linguagem ASM Concreta

O núcleo de cogen recebe uma representação abstrata de uma especificação ASM e produz uma representação abstrata de uma extensão de geração, como descrito na Seção 4.3. Para se trabalhar com uma linguagem ASM concreta, é necessária a construção de um "filtro". O filtro deve conter um analisador léxico e sintático da linguagem concreta, construindo uma representação abstrata a partir de uma representação textual. As rotinas do núcleo foram implementadas na linguagem C++, assim pelo menos parte do filtro deve ser também implementada em C++, para que possa gerar as classes que irão representar uma especificação ASM de entrada.

O núcleo processa apenas regras ASM básicas, mas as especificações de entrada podem ter construções mais complexas. Isso é possível se o filtro for capaz de traduzir essas construções para uma representação abstrata que utilize apenas regras básicas.

O filtro deve selecionar algumas partes do texto da especificação de entrada que não estão relacionadas às regras ASM processadas pelo núcleo, e copiá-las diretamente para a extensão de geração. Mas nesse caso todas as construções envolvidas deverão ser necessariamente estáticas. Por exemplo, a linguagem Xasm oferece facilidades para definir a gramática de uma linguagem L qualquer, gerando um analisador léxico e sintático para L. Todas as construções relacionadas à gramática devem ser copiadas diretamente para a extensão de geração.

Após o processamento executado pelo núcleo, a construção do texto da extensão de geração fica toda a cargo do filtro. Diferente do que acontecia com a especificação de entrada, a extensão de geração não precisa necessariamente utilizar apenas regras ASM básicas. Qualquer construção mais sofisticada oferecida pela linguagem concreta pode ser utilizada. No filtro que desenvolvemos para Xasm, por exemplo, foram utilizadas a execução de comandos seqüencialmente, abstração de funções e o comando "`do forall`", similar à regra ASM *var*.

```
asm GEN-TURING (progfile)

function _tapefile -> Int
function _tape -> Int
function _thead -> Int

function prog_instr(Int) -> Instr
function prog_par1(Int) -> Char
function prog_par2(Int) -> Int
function pc -> Int

function _TAB_pc(Int) -> Int
```

```
...
elseif prog_instr (pc) = IFGOTO then
  pc := prog_par2 (pc);
  SEARCHstate(1) := true;;
  pc := pc + 1
  SEARCHstate(2) := true;
  GenRESRULE (curS) :=
   GenCOND (
    "código residual para tape(thead) = prog_par1(pc)",
    GenGOTO (newstate(1)),
    GenGOTO (newstate(2)) )
```

Figura11. Trechos de um Compilador de MT para Xasm.

5 Experimentos com Cogen e Xasm

Todos os experimentos que conduzimos utilizaram `cogen` com um filtro para a linguagem Xasm. Nesta seção, apresentamos dois exemplos: o primeiro envolve o interpretador para máquina de Turing apresentado na Seção 2.3, e o segundo utiliza um interpretador para um significativo subconjunto da linguagem C.

5.1 Compilador para Máquina de Turing

O interpretador para uma versão da linguagem da máquina de Turing, cujo código é parcialmente exibido na Figura 1, foi submetido ao gerador de de extensões de geração `cogen`. Como é geralmente feito na especialização de interpretadores, o parâmetro de entrada `progfile`, que representa o nome do arquivo com o programa MT, foi definido como estático. O parâmetro de entrada `tapefile`, que representa o nome do arquivo com o estado inicial da fita, foi definido como dinâmico.

O resultado da aplicação de `cogen` a esse interpretador é um compilador da linguagem MT para Xasm. O cabeçalho e parte das declarações do compilador são exibidos na parte esquerda da Figura 11. Observe como o compilador tem apenas `progfile` como parâmetro de entrada. Funções inteiras são declaradas para armazenar identificadores únicos que serão associados a cada função dinâmica do interpretador. As funções estáticas são parte integrante do código do compilador, assim são declaradas da mesma forma que no interpretador. A única função estática que sofre atualização é `pc`. Assim uma função adicional `_TAB_pc`, indexada por um valor inteiro, é usada para armazenar todos os possíveis valores que `pc` pode assumir.

Um pequeno trecho de código da regra de transição do compilador é exibido no lado direito da Figura 11. Esse trecho está associado à geração de código residual para o comando `IFGOTO` da máquina de Turing. O sinal "`;`" indica execução seqüencial de regras em um programa Xasm, ao contrário da execução normal, que é paralela.

A função `curS` é um número inteiro que indica qual posição da tabela `_TAB_pc` contém um valor igual ao valor corrente de `pc`, que representa um estado estático. Observe que o valor de pc é alterado e em seguida há uma chamada de `SEARCHstate`. A função `SEARCHstate`, cujo código não é mostrado, executa uma série de procedimentos, que para o compilador são enxergados como executados em um único passo. Em uma chamada `SEARCHstate(`i`)`:

1. O valor de pc é pesquisado na tabela representada por __TAB_pc. Se não for encontrado, uma nova instância é adicionada a __TAB_pc, armazenando o valor de pc.
2. A função newstate(i) indicará a posição da tabela com o valor de pc.
3. A função pc recebe novamente o valor que possuía antes de ser alterado, ou seja, __TAB_pc(curS).

As ações descritas acima são exatamente o que havíamos discutido na Seção 4.3. Na Figura 11, são gerados dois possíveis novos estados estáticos. A função GenRESRULE associa cada estado estático a uma regra residual. Observe que os ramos da regra condicional residual gerada estão associados a cada um dos estados produzidos pelas atualizações estáticas, usando construtores *gotostate*.

A especificação residual deverá possuir uma função adicional curstate : *INT*, que define seu fluxo de controle. O valor inicial de curstate é 0 (zero). A regra de transição residual é um bloco de regras da forma "if curstate = k then R_k", para cada valor k do domínio de GenRESRULE. O fluxo de controle é determinado por atualizações sobre curstate, as quais são geradas pelas chamadas a GenGOTO.

5.2 Compilador para um Subconjunto da Linguagem C

Utilizando idéias semelhantes às apresentadas em [11], escrevemos, em Xasm, um interpretador para um significativo subconjunto da linguagem C. É possível submeter à interpretação programas em C com:

- expressões com operadores aritméticos básicos, auto-incremento e auto-decremento;
- declarações de variáveis em blocos e chamadas recursivas de funções;
- comando de seleção IF e comando de repetição WHILE;
- declarações e uso de registros e apontadores;
- alocação de memória dinâmica.

Como em [11], o programa C a ser interpretado é visto como um grafo, onde a cada passo um nodo é visitado. Uma função CurTask indica qual é a instrução corrente. A cada nodo do grafo estão associadas regras que são executadas, juntamente com a definição do novo valor de CurTask. Para utilizar essa representação, o programa fonte em C é traduzido para uma forma intermediária, por meio de ações associadas às regras de uma gramática definida dentro do texto da especificação ASM.

BTA e Chamadas Recursivas de Funções Um problema muito comum na especialização de interpretadores para linguagens com chamadas recursivas de funções teve que ser enfrentado. O interpretador possui duas entradas: o texto do programa P a ser interpretado e os dados de entrada para P. A extensão de geração será produzida tendo P como entrada estática e os dados de P como entrada dinâmica. Para obter bons resultados na especialização, é imprescindível ter a função CurTask, que indica a instrução corrente sendo interpretada, classificada como estática.

No interpretador de C desenvolvido, uma pilha é utilizada para armazenar valores de computações intermediárias e também informações necessárias para chamadas recursivas de funções, como por exemplo o ponto de retorno no código. Como armazena valores que dependem dos dados do programa P sendo interpretado, essa pilha deve ser classificada como dinâmica. Entretanto, numa chamada de uma função f, o valor de CurTask também

Interpretador de C escrito em Xasm	
Geração do Executável com `XasmC`	12s
Execução de P_1	26s
Execução de P_2	38s

Compilador de C para Xasm (CtoXasm)		
Geração do Compilador com `cogen`		7s
Geração do Executável com `XasmC`		18s
	Programa P_1	Programa P_2
Compilação com `CtoXasm`	10s	53s
Executável com `XasmC`	9s	14s
Execução	15s	22s
Ganho em Velocidade	42%	42%

Figura12. Testes com o Interpretador de C.

é armazenado na pilha, indicando o ponto de retorno após a execução de f. No retorno de f, `CurTask` é atualizada com o valor armazenado. De acordo com o algoritmo apresentado na Seção 4.2, CurTask deveria ser classificada como dinâmica, o que iria produzir resultados muito fracos na especialização.

Existe uma forma simples de contornar esse problema, conhecida como "O Truque" [15]. Analisando o código do programa P, que é uma informação estática, é fácil determinar os possíveis valores assumidos por `CurTask` no retorno de cada função. Na tradução para a forma intermediária, nos pontos em que uma função f retorna, são inseridas instruções do tipo "`return` $\mathtt{r}_i$", uma para cada possível ponto de retorno $\mathtt{r}_i$ de f. Uma regra no formato "`if Stack(index) =` $\mathtt{r}_i$ `then CurTask :=` $\mathtt{r}_i$ `endif`" estabelece a semântica dessas instruções no interpretador de C, onde `index` indica a posição na pilha onde o ponto de retorno está armazenado. Observe que o teste da regra condicional é dinâmico, mas `CurTask` pode continuar sendo classificada como estática.

Testes A Figura 12 mostra medidas de tempo realizadas em alguns testes conduzidos sobre o interpretador de C e um compilador gerado a partir do mesmo. As medidas são uma média de 5 aferições, usando uma estação SUN UltraSPARC 2. O ponto mais interessante é a comparação entre a execução dos programas interpretados e compilados.

Primeiramente, o interpretador de C, escrito em Xasm, foi transformado em código executável usando o compilador de Xasm disponível, chamado de `XasmC`. No lado esquerdo da Figura 12, pode-se ver medidas de tempo associadas à interpretação de dois programas escritos em C. O programa P_1 gera um trecho da seqüência de Fibonnacci, com uso intenso de recursividade. Foi executado para gerar o décimo-quarto número da seqüência. O programa P_2 ordena um vetor de inteiros, usando o método ingênuo de ordenação por seleção. Foi executado com um conjunto de 30 inteiros.

Em seguida, um compilador de C para Xasm foi produzido, usando o gerador de extensões de geração `cogen` desenvolvido. Os mesmos programas P_1 e P_2 foram então compilados para Xasm e depois um executável foi gerado usando `XasmC`. No lado direito da Figura 12, são exibidas medidas de tempo para essas tarefas e para a execução dos programas compilados.

6 Conclusão

Técnicas de avaliação parcial têm como objetivo principal a melhoria de desempenho de programas, produzindo versões especializadas que executam mais rápido que os programas originais, mais genéricos. Uma área onde essas técnicas têm sido utilizadas com sucesso

é na geração de compiladores dirigida por semântica [14, 16, 17]. Um gerador de compiladores pode ser automaticamente produzido por meio da auto-aplicação de um avaliador parcial, mas existem casos em que a auto-aplicação não é muito adequada. Nesses casos, a abordagem mais indicada é escrever o gerador de compiladores à mão. Essa técnica é conhecida como abordagem de extensões de geração.

Neste trabalho, apresentamos um gerador de extensões de geração `cogen` para a linguagem ASM. Se a especificação ASM submetida a `cogen` é um interpretador de uma linguagem L, então um compilador de L para ASM é automaticamente gerado. Discutimos razões que nos levaram a escrever `cogen` à mão, em vez de gerá-lo automaticamente por meio da auto-aplicação de um avaliador parcial tradicional.

Exibimos os algoritmos que implementam o núcleo do gerador de extensões de geração usando o próprio modelo ASM, juntamente com definições de funções em uma linguagem funcional estrita. As rotinas do núcleo podem ser utilizadas com qualquer linguagem ASM concreta, se um filtro que implemente uma interface com o núcleo estiver disponível. Implementamos um filtro para a linguagem Xasm e realizamos alguns testes, sendo o mais significativo a geração de um compilador para um subconjunto da linguagem C.

O objetivo principal do trabalho é melhorar o desempenho na execução de especificações ASM, tornando assim mais atraente o uso desse modelo. Os resultados exibidos na Seção 5.2 mostram melhoras no tempo de execução de programas escritos em C, usando um compilador de C para Xasm automaticamente gerado. Consideramos que o fato de Xasm não ser uma linguagem fortemente tipada seja o principal motivo para essas melhoras não serem mais significativas.

Nossos planos futuros incluem a implementação de diversas melhorias no programa `cogen`, possibilitando a geração de código mais eficiente e o processamento de regras ASM mais complexas. Incluem também a construção de um filtro para outra linguagem que usa o modelo ASM, designada Machĭna[19]. Como é uma linguagem fortemente tipada, acreditamos que o uso de `cogen` irá produzir melhoras mais significativas para o tempo de execução de especificações compiladas, quando comparado à interpretação das mesmas.

Referências

1. L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.
2. M. Anlauff. Xasm – An Extensible, Component-Based Abstract State Machines Language. In *Proceedings of the ASM 2000 Workshop*, pages 1–21, Monte Veritá, Switzerland, March 2000.
3. L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September, 1994. (Lecture Notes in Computer Science, Vol. 844)*, pages 198–214. Berlin: Springer-Verlag, 1994.
4. E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J. Bowen, M. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1997.
5. 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.
6. 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.
7. V. O. Di Iorio and R. S. Bigonha. Avaliação Parcial de Máquinas de Estado Abstratas. In *Anais do III Simpósio Brasileiro de Linguagens de Programação*, pages 45–59, Porto Alegre, Maio 1999.
8. V. O. Di Iorio, R. S. Bigonha, and M. A. Maia. A Self-Applicable Partial Evaluator for ASM. In *Proceedings of the ASM 2000 Workshop*, pages 115–130, Monte Veritá, Switzerland, March 2000.

9. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
10. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
11. 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.
12. 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.
13. 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.
14. R. Heldal. Generating more practical compilers by partial evaluation. In R. Heldal, C. Kehler Holst, and P. Wadler, editors, *Functional Programming, Glasgow 1991*, pages 158–163. Berlin: Springer-Verlag, 1992.
15. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.
16. N. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
17. J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992*, pages 258–268. New York: ACM, 1992.
18. J. Launchbury and C. Holst. Handwriting cogen to avoid problems with static typing. In *Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218, 1991.
19. F. Tirelo, R. Bigonha, M. A. Maia, and V. Iorio. Machĭna: A Linguagem de Especificação de ASM (in portuguese). Technical Report 08/1999, Laboratório de Linguagens de Programação, Universidade Federal de Minas Gerais, 1999.