1 Introdução

As Máquinas de Estado Abstratas (ASM) são um formalismo criado por Yuri Gurevich [5], originalmente conhecido por *Evolving Algebras*. É utilizado 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 com respeito 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` [3].

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 [4] no início da década de 70, mas só foram realizados com sucesso pela primeira vez em meados dos anos 80 [8].

Neste artigo descrevemos as técnicas utilizadas para desenvolver um avaliador parcial `mix` 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. 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 [6], mas não permitia a auto-aplicação e conseqüentemente não possibilitava a geração de compiladores.

O artigo está organizado da forma descrita a seguir. Na seção 2, o formalismo das Máquinas de Estado Abstratas é apresentado, concentrando-se nos aspectos relativos ao seu uso para descrição da semântica de linguagens de programação. A seção 3 discute aspectos relacionados à avaliação parcial de programas. As técnicas utilizadas para desenvolver um avaliador parcial para ASM são discutidas na seção 4. A geração de compiladores usando o avaliador parcial para ASM é o assunto da seção 5. A seção 6 apresenta as conclusões e futuros trabalhos.

2 ASM Seqüenciais

As Máquinas de Estado Abstratas constituem um formalismo poderoso o bastante para especificar de maneira simples algoritmos envolvendo paralelismo e processamento distribuído. Entretanto, nosso principal objetivo envolve sua utilização na especificação da semântica de linguagens de programação, e por enquanto nos limitamos a linguagens para processamento seqüencial.

Esta seção descreve o subconjunto das ASM designado por *ASM Seqüenciais*, adequado aos nossos propósitos. As definições visam tornar o artigo autocontido, mas uma abordagem mais completa e formal sobre ASM pode ser encontrada em [5] e [2].

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 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 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 *instrução de atualização*, *construtor de bloco* ou *construtor condicional*.

Uma instrução de atualização tem o formato $f(\bar{t}) := t_0$, 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 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 g_0 then R_0
elseif g_1 then R_1
⋮
elseif g_k then R_k
endif
```

A semântica do comando condicional é a seguinte: uma regra R_i, $0 \leq i \leq k$, será 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. A implementação utilizada para conduzir os experimentos divulgados neste artigo utiliza um novo comando `stop`, cuja semântica é indicar o final da execução do programa.

2.2 Um exemplo simples: seqüência de Fibonacci

O exemplo a seguir representa um algoritmo para computar os 100 primeiros valores da seqüência de Fibonacci.

```
Inicialização:
    fib(1) := 0
    fib(2) := 1
    ind := 3
Regras:
    IF ind > 100 THEN
        STOP
    ELSE BEGIN
        ind := ind + 1
        fib(ind) := fib(ind-1) + fib(ind-2)
    END
```

O programa possui uma seção que determina o valor das funções no estado inicial (aqui designada *inicialização*) e outra que determina a regra de transição (designada *regras*). Observe o funcionamento da atualização de uma função, característica incomum em linguagens de programação. O estado inicial é caracterizado por $\{ind = 3, fib(1) = 0, fib(2) = 1\}$. Após a primeira transição, o estado será $\{ind = 4, fib(1) = 0, fib(2) = 1, fib(3) = 1\}$. Os delimitadores `BEGIN...END` são utilizados para determinar um bloco; observe que a ordem das regras dentro de um bloco é irrelevante, uma vez que as atualizações são realizadas em paralelo.

2.3 Especificando um interpretador

Este exemplo exibe um interpretador de uma versão da *Máquina de Turing*. A máquina possui as seguintes instruções: `right`, `left`, `write a`, `goto i`, `if a goto i`, `stop`. Um estado é caracterizado pela instrução I_i que será executada no próximo passo, juntamente com uma fita infinita cujas células podem armazenar elementos do conjunto $\{0, 1, \texttt{undef}\}$ e um valor que indica a posição da cabeça de leitura/gravação, ou seja, qual célula da fita está sendo analisada no momento. Apenas um número finito de células da fita possui valor diferente de `undef` e inicialmente a cabeça indica a primeira célula com essa característica, se houver.

A instrução `write a` altera o conteúdo da célula analisada para 0, 1 ou `undef`, conforme o valor de `a`; `right` desloca a cabeça uma célula para a direita; `left` desloca a cabeça uma célula para a esquerda; `goto i` desvia o fluxo do programa para a instrução I_i; `if a goto i` é um desvio que só ocorre se o conteúdo da célula analisada for `a`; `stop` termina a execução do programa. O programa exemplo apresentado a seguir, extraído de [7], pode provocar uma alteração na fita ou entrar em *loop* infinito, se a fita não contiver nenhum símbolo 0:

```
0:  if 0 goto 3
1:  right
2:  goto 0
3:  write 1
4:  stop
```

Apresentamos em seguida uma regra de transição ASM que implementa um interpretador para a MT descrita acima. Vamos supor que a fita seja representada pela função `fita`, inicializada de forma adequada. Vamos supor também que o programa pode ser recuperado por meio das funções `cod`, `par1` e `par2` que, dado o número de uma instrução, fornecem respectivamente o código, valor do primeiro parâmetro e valor do segundo parâmetro dessa instrução. O número da instrução corrente é indicado por `cont`, e a célula da fita analisada é indicada por `cabeca`. O código ASM é exibido abaixo:

```
IF cod(cont) = 'GOTO' THEN
    cont := par1(cont)
IF cod(cont) = 'IFGOTO' THEN
    IF par1(cont) = fita(cabeca) THEN
        cont := par2(cont)
    ELSE
        cont := cont + 1
IF cod(cont) = 'WRITE' THEN BEGIN
    cont := cont + 1
    fita(cabeca) := par1(cont)
END
IF cod(cont) = 'LEFT' THEN BEGIN
    cont := cont + 1
    cabeca := cabeca - 1
END
IF cod(cont) = 'RIGHT' THEN BEGIN
    cont := cont + 1
    cabeca := cabeca + 1
END
IF cod(cont) = 'STOP'
    STOP
```

3 Avaliação Parcial

Um *avaliador parcial* é um programa que 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 a entrada dinâmica de P, produz os mesmos resultados que P produziria se fosse executado com ambas as entradas.

3.1 Exemplo

Como exemplo simples, observe a função *Power*, escrita na linguagem C:

```
Int Power (int n, int x) {
    int p = 1;
    while (n > 0) {
        if (n%2 == 0) { x = x * x; n = n / 2; }
        else { p = p * x; n = n - 1; }
    }
    return p;
}
```

A especialização de *Power* dado $n = 5$ é uma função com apenas um parâmetro x. Essa função deve produzir o mesmo resultado que *Power*, se for executada com entradas 5 e x, para qualquer valor de x. Uma solução ingênua, mas que satisfaz a condição imposta acima, seria:

```
int Power_5 (int x) {
    return Power (5, x);
}
```

De fato, o *Teorema s-m-n* de Kleene [7] mostra que sempre é possível obter um programa especializado, e sua prova exibe o projeto de um avaliador parcial. Esse teorema, entretanto, não se preocupa com questões de eficiência. No exemplo em questão, o valor estático n permite obter um código mais eficiente, como o exibido a seguir:

```
int Power_5 (int x) {
    int p = x;
    x = x * x; x = x * x;
    p = p * x;
    return p;
}
```

3.2 Geração de geradores de programas

Seguindo a notação utilizada em [7], se P é um programa escrito na linguagem L, $[\![P]\!]_L$ 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:

$$\begin{aligned} out &= [\![P]\!]_S\,(in_1, in_2) \\ P_{in_1} &= [\![mix]\!]_L\,(P, in_1) \\ out &= [\![P_{in_1}]\!]_T\,(in_2) \end{aligned}$$

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* [4]. 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:

$$\begin{aligned} out &= [\![source]\!]\,(input) \\ &= [\![int]\!]\,(source, input) \\ &= [\![[\![mix]\!]\,(int, source)\,]\!]\,(input) \\ &= [\![target]\!]\,(input) \end{aligned}$$

Assim temos $target = [\![mix]\!]\,(int, source)$, 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`:

$$\begin{aligned} target &= [\![mix]\!]\,(int, source) \\ &= [\![[\![mix]\!]\,(mix, int)\,]\!]\,(source) \\ &= [\![compiler]\!]\,(source) \end{aligned}$$

Temos então $compiler = [\![mix]\!]\,(mix, int)$. 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. A terceira projeção de Futamura envolve geração de geradores de compiladores:

$$\begin{aligned} compiler &= [\![mix]\!]\,(mix, int) \\ &= [\![[\![mix]\!]\,(mix, mix)\,]\!]\,(int) \\ &= [\![cogen]\!]\,(int) \end{aligned}$$

O primeiro avaliador parcial auto-aplicável foi construído por Jones, Sestoft e Sondergaard, para uma linguagem de equações recursivas de primeira ordem. A primeira versão [8] requeria anotações prévias introduzidas pelo usuário, mas uma versão seguinte [9] era completamente automática. Experimentos de Jorgensen [10] mostraram ser possível obter, através dos métodos discutidos acima, um código compilado com velocidade de execução equivalente ao produzido por compiladores comerciais.

3.3 Métodos *online* e *offline*

As técnicas utilizadas para avaliação parcial podem ser divididas em abordagens *online* e *offline*. Os primeiros avaliadores parciais usavam sempre métodos *online*.

mas as técnicas *offline* parecem ser necessárias para permitir auto-aplicação e a geração de geradores de programas [7].

Todo avaliador parcial recebe como entradas um programa P a ser especializado, juntamente com a indicação de que parte da entrada de P é estática e qual é o valor dessa entrada estática. Todos os objetos referenciados no texto do programa P são classificados como estáticos ou dinâmicos, dependendo de sua relação com os valores estáticos e dinâmicos da entrada de P. O avaliador executa trechos de código associados aos valores estáticos e gera código para os trechos associados aos valores dinâmicos.

Nas técnicas *online*, a avaliação parcial é geralmente executada em uma única fase, e os valores são definidos como estáticos ou dinâmicos durante a especialização. Os métodos *offline* consistem geralmente em duas fases: a primeira fase é chamada de BTA (*binding time analysis* ou *análise de tempo de definição*) e gera um programa anotado que indica o caráter estático ou dinâmico de cada objeto e operação; a segunda fase (especialização) segue essas anotações para produzir o programa especializado.

Um fator negativo dos métodos *online* é a falta de uma análise global prévia do programa a ser especializado. Eles permitem, em alguns casos, um resultado mais eficiente por terem acesso a informações durante o processo de especialização não disponíveis aos métodos *offline*. Entretanto sofrem de uma maior dificuldade com relação a assegurar a terminação do processo. Os métodos *offline* facilitam a auto-aplicação exatamente por dividir o processamento em duas fases. O código do avaliador parcial que é submetido a si próprio consiste em um programa anotado mais simples, pois executa apenas a segunda fase do método (especialização).

4 Avaliador Parcial para ASM

O primeiro avaliador parcial construído para ASM que se tem notícia foi apresentado por Gurevich e Huggins em [6] e utilizava técnicas *offline*. Não permitia entretanto a auto-aplicação, impossibilitando verificar os resultados da aplicação da segunda e terceira projeções de Futamura. O trabalho apresentado neste artigo também utiliza métodos *offline* e permite a geração de compiladores usando a auto-aplicação, assunto abordado na seção 5. Estudos estão sendo conduzidos para obter a geração de geradores de compiladores usando o avaliador parcial.

4.1 Análise de Tempo de Definição

Análise de tempo de definição, ou BTA (*binding time analysis*), é a primeira fase da avaliação parcial usando métodos *offline* e consiste em classificar cada objeto e operação como estático ou dinâmico. Os nomes "dinâmico" e "estático" têm outra conotação em ASM, sendo atribuídos, respectivamente, a funções que podem ou não sofrer atualizações. Seguindo a sugestão de Huggins, vamos utilizar o termo "negativo" no lugar de "dinâmico", e "positivo" no lugar de "estático".

A entrada para um programa ASM P é definida por um conjunto de funções cujo valor é fornecido pelo usuário. O avaliador parcial recebe uma indicação de quais dessas funções de entrada são positivas e quais são negativas. O primeiro passo da BTA é classificar todas as outras funções referenciadas em P.

O algoritmo funciona da forma descrita a seguir. Inicialmente, as funções que não fazem parte da entrada não são classificadas como positivas nem negativas. A cada iteração, uma função f será classificada como negativa se existir uma atualização $f\left(\bar{t}\right) := t_0$ onde $\bar{t}$ ou t_0 referenciam uma função negativa. O processo é repetido até que seja alcançado um ponto fixo. As funções não classificadas como negativas são a princípio positivas, mas podem também ser classificadas como negativas em alguns casos. Um exemplo onde isso acontece é o seguinte:

```
IF y ≠ 0 THEN BEGIN
    x := x + 1; y := y - 1
END
```

Supondo que x seja positivo (estático) e y negativo (dinâmico), os valores que x assume não são limitados por uma condição estática. Isso pode levar o avaliador parcial a um *loop* infinito. Nesse caso, x deve ser classificado como negativo. Outros casos devem ser analisados, levando sempre em consideração se a função poderá assumir um número infinito de valores diferentes.

A BTA deve determinar um conjunto mínimo de funções negativas, de modo a maximizar a especialização. Entretanto, deve fazê-lo de modo a garantir que o avaliador parcial não entre em loop. Esse problema não é computável [7]. O avaliador parcial construído procura implementar uma solução aproximada, identificando alguns dos casos onde as funções devem ser obrigatoriamente negativas, e permite também que o usuário classifique qualquer função como negativa através de anotações.

Depois de classificar todas as funções referenciadas em P, cada operação e comando é também classificado. Um termo $f\left(\bar{t}\right)$ será negativo se f for negativa ou se algum dos termos de $\bar{t}$ for negativo. Uma atualização $f\left(\bar{t}\right) := t_0$ será negativa se f foi classificada como negativa. Finalmente, um comando condicional será negativo se a condição testada for negativa. Em todos os outros casos, essas construções serão classificadas como positivas, gerando um programa anotado designado por P_1.

No exemplo da seção 2.3, suponha que o interpretador será avaliado parcialmente com respeito ao programa fonte exibido na mesma seção. Supondo que o programa inicie pela instrução 0, as funções `cont`, `cod`, `par1` e `par2` serão classificadas como positivas, enquanto `fita` e `cabeca` serão negativas. Os comandos de atualização que envolvem as funções `fita` e `cabeca` serão negativos, sendo todos os outros positivos. Apenas uma condição de um comando `if` será negativa (observe o código completo na seção seguinte).

4.2 Pré-Processamento

Huggins propõe um pré-processamento do programa de modo a poder simplificar o código do avaliador parcial. Esse pré-processamento é realizado antes da fase de BTA e pode fazer o programa crescer exponencialmente. A implementação que desenvolvemos utiliza transformações semelhantes, mas elas são conduzidas após a fase de BTA, procurando minimizar a expansão do código do programa.

Em cada bloco, as regras são reordenadas de modo a posicionar os comandos condicionais no final do mesmo. Se existirem dois comandos condicionais no bloco, o código do último é eliminado e inserido simultaneamente na cláusula THEN e

na cláusula ELSE do penúltimo. Esse processo é recursivo e se repete até que o bloco contenha no máximo um comando condicional. No último nível da recursão, é acrescentado em cada bloco um comando especial dummy que será usado na fase de especialização como indicador da geração de um novo estado (explicado mais adiante). Apenas os blocos que contenham stop não terão um comando dummy, pois não irão gerar novos estados.

O processo utilizado gera uma expansão de código menor do que a proposta por Huggins, porque os comandos que não dependem de avaliação de condições não são duplicados dentro do mesmo bloco. Além disso, utiliza o programa anotado P_1 para identificar condições positivas, evitando a duplicação também nesse caso. O novo programa, anotado e pré-processado, é designado P_2.

O resultado da aplicação da BTA e do pré-processamento ao exemplo da seção 2.3 é exibido a seguir. Um sinal junto de cada estrutura indica se a mesma foi classificada como positiva(⊕) ou negativa(⊖). Observe que o processo de expansão do código descrito acima, com a eliminação de comandos if, não foi aplicado nesse caso. A BTA permitiu verificar que os comandos condicionais eram positivos, o que elimina a necessidade da expansão.

```
⊕IF ⊕cod(⊕cont) ⊕= ⊕''GOTO'' THEN BEGIN
    ⊕cont ⊕:= ⊕par1(⊕cont)
    DUMMY
END
⊕IF ⊕cod(⊕cont) ⊕= ⊕''IFGOTO'' THEN
    ⊖IF ⊕par1(⊕cont) ⊖= ⊖fita(⊖cabeca) THEN BEGIN
        ⊕cont ⊕:= ⊕par2(⊕cont)
        DUMMY
    END
    ELSE BEGIN
        ⊕cont ⊕:= ⊕cont ⊕+ ⊕1
        DUMMY
    END
⊕IF ⊕cod(⊕cont) ⊕= ⊕''WRITE'' THEN BEGIN
    ⊕cont ⊕:= ⊕cont ⊕+ ⊕1
    ⊖fita(⊖cabeca) ⊖:= ⊕par1(⊕cont)
    DUMMY
END
⊕IF ⊕cod(⊕cont) ⊕= ⊕''LEFT'' THEN BEGIN
    ⊕cont ⊕:= ⊕cont ⊕+ ⊕1
    ⊖cabeca ⊖:= ⊖cabeca ⊖- ⊕1
    DUMMY
END
⊕IF ⊕cod(⊕cont) ⊕= ⊕''RIGHT'' THEN BEGIN
    ⊕cont ⊕:= ⊕cont ⊕+ ⊕1
    ⊖cabeca ⊖:= ⊖cabeca ⊖+ ⊕1
    DUMMY
END
⊕IF ⊕cod(⊕cont) ⊕= ⊕''STOP''
    STOP
```


4.3 Especialização

A especialização é conduzida sobre o programa P_2 produzido na fase anterior, que determinou também o conjunto de funções classificadas como positivas, designado por Fp. O avaliador parcial trabalha sobre uma ASM $\mathcal{A}_{Fp}$ cuja assinatura está restrita a Fp e procura determinar todos os possíveis estados alcançados a partir do estado inicial.

O processamento começa com o estado inicial de $\mathcal{A}_{Fp}$. A regra do programa P_2 é analisada, sendo que trechos de código são gerados para cada estrutura negativa e as atualizações positivas dão origem a novos estados. Cada novo estado é analisado, produzindo um código associado e gerando mais estados, que podem ou não já terem sido analisados. Esse procedimento termina quando todos os possíveis estados foram analisados. A forma como a BTA foi conduzida assegura a terminação do processo, a não ser que o programa original possua um loop infinito gerado pelas funções positivas.

Para produzir o código associado a um estado S e o conjunto de novos estados, a regra do programa P_2 é processada da seguinte maneira:

- Se a regra for um bloco, cada comando é processado.
- Um comando de atualização positivo gera uma atualização no estado corrente.
- Um comando de atualização negativo $f\left(\bar{t}\right) := t_0$ produz como código uma atualização com todas as estruturas positivas de $\bar{t}$ e t_0 avaliadas.
- Em um comando condicional positivo, primeiro a condição é avaliada. Se for verdadeira, processa-se a cláusula THEN; senão, processa-se a cláusula ELSE.
- Um comando condicional negativo produz como código um comando condicional com as estruturas positivas da condição avaliadas e o processamento das cláusulas THEN e ELSE. Neste caso, entretanto, o estado corrente é duplicado, gerando uma cópia que será atualizada pelo processamento da cláusula THEN e outra para a cláusula ELSE.
- Os comandos dummy inseridos na fase anterior indicam os pontos em que todas as atualizações do estado corrente já foram processadas. Ao encontrar um dummy, as atualizações acumuladas são processadas sobre o estado corrente, em paralelo, e um (novo) estado é gerado. O código produzido deve ser uma construção que referencie o estado gerado, informação que será utilizada para gerar o programa residual final.

Para exemplificar esse processo, suponha o programa anotado P_2 exibido na seção anterior, avaliado parcialmente com respeito ao programa exemplo MT da seção 2.3. O primeiro comando do programa MT é if 0 goto 3; supondo que o valor inicial de cont seja 0, o código gerado na primeira iteração terá o formato seguinte:

```
IF 0 = fita(cabeca) THEN
    DUMMY{cont=3}
ELSE
    DUMMY{cont=1}
```

Os comandos dummy indicam os novos estados gerados. Nesse caso, devem ser gerados estados onde o valor de cont é 3 e 1. Como estabelecido acima, a presença de um comando if negativo foi responsável pela duplicação do estado. Na realidade, os estados gerados são representados pelos valores de todas as funções positivas, mas apenas cont é necessariamente representada por ser a única que sofre atualização.

A ordem de execução dos comandos no programa residual é definida por esse processo. Na regra acima, por exemplo, o código que deverá ser executado caso a condição seja verdadeira será aquele gerado pela avaliação parcial de P_2 com cont atualizado para 3. O comando dummy corresponderia a um goto em uma linguagem imperativa comum.

Ao fim do processo, temos um conjunto de regras $\{R_0, \ldots, R_k\}$, uma para cada possível estado, supondo $k+1$ estados. Suponha que R_0 é um conjunto de atualizações que construa o estado inicial de $\mathcal{A}_{Fp}$. Adiciona-se ao superuniverso de $\mathcal{A}_{Fp}$ um novo elemento distinto, designado pe_flag. Cada comando dummy é substituído pela atualização pe_flag := i, onde i é o estado referenciado por esse comando, conforme estabelecido acima. O programa residual terá o seguinte formato:

```
Inicialização:
    pe_flag := 0
Regras:
    IF pe_flag = 0 THEN R0
    ...
    IF pe_flag = k THEN Rk
```

Voltando ao exemplo da seção 2.3, o programa residual é apresentado a seguir.

```
Inicialização:
    pe_flag := 0
Regras:
    IF pe_flag = 0 THEN BEGIN
        {... valores iniciais de cabeca e fita...}
        pe_flag := 1
    END
    IF pe_flag = 1 THEN
        IF fita(cabeca) = 0 THEN pe_flag := 2
        ELSE pe_flag := 3
    IF pe_flag = 2 THEN BEGIN
        fita(cabeca) := 1; pe_flag := 5
    END
    IF pe_flag = 3 THEN BEGIN
        cabeca := cabeca + 1; pe_flag := 4
    END
    IF pe_flag = 4 THEN pe_flag := 1
    IF pe_flag = 5 THEN stop
```

Uma série de otimizações devem ser implementadas de modo a tornar o programa residual mais eficiente. Uma otimização de fluxo de controle simples é conhecida por *transition compression* [1], que em outras linguagens imperativas significaria eliminação de gotos redundantes. Essa situação é representada por uma regra da forma IF pe_flag = i THEN pe_flag := j.

As regras geradas que serão executadas em seqüência, mas cujas atualizações não são conflitantes, também podem ser reunidas em um único bloco. Essas duas otimizações, entre outras, foram implementadas no avaliador parcial construído. A compressão e eliminação de regras é realizada durante o processo de especialização (*transition compression on the fly*), ao invés de ser feita numa fase posterior. Mesmo sendo uma solução de mais difícil implementação, ela é preferível nesse caso devido ao alto número de regras redundantes geradas pelo processo.

Se forem aplicadas as técnicas de *transition compression* discutidas acima, o resultado será o seguinte:

```
Inicialização:
    {... valores iniciais de cabeca e fita...}
Regras:
    IF fita(cabeca) = 0 THEN BEGIN
        fita(cabeca) := 1
        STOP
    END
    ELSE cabeca := cabeca + 1
```

Observe que o programa residual possui exatamente a mesma semântica que o programa MT do exemplo. Realizou-se uma compilação de MT para ASM via avaliação parcial de um interpretador MT, com respeito a um programa fonte específico. Como o interpretador define a semântica de MT, temos *compilação dirigida por semântica.*

5 Geração de Compiladores

A segunda projeção de Futamura, como visto na seção 3.2, refere-se à auto-aplicação do avaliador mix para gerar compiladores a partir de interpretadores. Naquela seção, designamos L como a linguagem em que mix é escrito e S como a linguagem dos programas submetidos ao avaliador parcial. Vimos que, para satisfazer a segunda projeção de Futamura, é necessário ter mix escrito em S.

O avaliador parcial divulgado neste artigo foi escrito em Java, aplicado a programas escritos na linguagem ASM. Vamos designar esse avaliador parcial por $\texttt{mix}_1$. Para obter a geração de compiladores dirigida por semântica, uma nova versão $\texttt{mix}_2$ foi escrita na linguagem ASM, de modo a poder servir de entrada para $\texttt{mix}_1$. O avaliador $\texttt{mix}_2$ não é tão poderoso quanto $\texttt{mix}_1$, implementando apenas a fase de especialização do método *offline*. A simplificação de $\texttt{mix}_2$ é uma caracterísitica bastante desejável nesse caso, pois facilita e torna mais rápida a geração de um compilador.

Como realiza apenas a fase de especialização, a entrada para $\texttt{mix}_2$ é um programa anotado P_{an} e o valor das entradas estáticas de P_{an}. As anotações estabelecem as estruturas positivas e negativas, definidas por uma BTA executada previamente. Além disso, o programa P_{an} é fornecido no formato de uma árvore de sintaxe abstrata, de modo que $\texttt{mix}_2$ não tenha que se preocupar com questões relativas a sintaxe. O processo de geração do programa anotado foi facilmente implementado usando as próprias estruturas do avaliador parcial original $\texttt{mix}_1$.

Temos então $compiler = [\![mix_1]\!]\,(mix_2, int_{an})$. O compilador gerado pela aplicação ao interpretador da seção 2.3 ficou bastante extenso (mais de 10 vezes o tamanho do interpretador). A qualidade do código gerado por esse compilador, quando comparada à obtida na seção anterior (compilação via avaliação parcial de um interpretador usando $\texttt{mix}_1$), perde apenas por não implementar *transition compression*. Essa otimização ainda não foi implementada em $\texttt{mix}_2$. Em média, compilação utilizando o compilador gerado pela equação acima foi três vezes mais rápida do que a compilação via avaliação parcial.

6 Conclusões e Futuros Trabalhos

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, como foi mostrado no exemplo da seção 4. Resultados semelhantes estão sendo obtidos com descrições de semântica de linguagens mais complexas, em especial um subconjunto da linguagem Java.

Consideramos a adoção de ASM para especificar a semântica de linguagens um ponto positivo do trabalho. A linguagem é simples e utiliza o paradigma imperativo, o que a torna acessível a um número maior de usuários, se comparada a uma abordagem com paradigma funcional. Na realidade, a linguagem ainda permite especificar o código de funções de modo similar ao paradigma funcional, combinando as duas abordagens. Um ponto negativo é que a compilação via avaliação parcial leva necessariamente à geração de código também na linguagem ASM, que ainda não possui implementações muito eficientes disponíveis.

Um teste é sugerido em [7] para verificar se um avaliador parcial atinge um desempenho satisfatório. O teste consiste em escrever um meta-interpretador para a linguagem S (programas submetidos a `mix`) e avaliá-lo parcialmente com respeito a programas escritos em S. O avaliador parcial deve ser capaz de eliminar todo o *overhead* de interpretação, dando como resultado um programa equivalente ao fornecido como entrada para o meta-interpretador. Para este trabalho, foi construído um meta-interpretador para ASM, e os resultados alcançados satisfizeram os critérios impostos pelo teste.

Com 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 $\texttt{mix}_2$ as otimizações presentes em $\texttt{mix}_1$. Em seguida, compiladores para linguagens mais complexas serão gerados e avaliados.

A aplicação da terceira projeção de Futamura, relacionada à geração de geradores de compiladores, ainda não foi testada. 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* [5], *ASM Distribuída* [5] e *ASM Interativa* [12, 11] permitem exprimir esses conceitos de forma bastante simples e direta.

Referências

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[2] 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.

[3] A. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[4] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

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

[6] 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.

[7] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

Avaliação Parcial de Máquinas de Estado Abstratas

Vladimir O. Di Iorio* Roberto S. Bigonha†

Resumo As *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. 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 artigo divulgamos as técnicas usadas para desenvolver 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. Finalmente, um compilador de L para ASM pode ser automaticamente gerado via auto-aplicação do avaliador parcial.
Palavras-Chave: Máquinas de Estado Abstratas, avaliação parcial, geração de compiladores dirigida por semântica.

Abstract The *Abstract State Machines* (ASM, for short) are a formalism created to model algorithms at its natural abstraction level. The semantics of a programming language can be described in ASM by defining an interpreter for the language. A *partial evaluator*, when given a subject program and part of its input, produces a new residual or specialized program. The residual program, when given the remaining input data, will yield the same result that the original program would have produced given both inputs. In this work we show the techniques used to develop a partial evaluator for the ASM language. Compiling from any language to ASM can be achieved by the partial evaluation of an interpreter. Compilers can also be automatically generated by self-application of the partial evaluator.
Key words: Abstract State Machines, partial evaluation, semantics-directed compiler generation.

*Email: vladimir@dcc.ufmg.br
Departamento de Informática - Universidade Federal de Viçosa
Campus Universitário - 36571-000 Viçosa - MG
†Email: bigonha@dcc.ufmg.br
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Av. Antônio Carlos 6627 - Prédio do Icex - 31270-010 Belo Horizonte - MG