Vladimir Oliveira Di Iorio

Avaliação Parcial em Máquinas de Estado Abstratas

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Belo Horizonte

28 de março de 2001

Agradecimentos

Gostaria de agradecer ao Professor Roberto da Silva Bigonha, entre outras coisas, por sua orientação e por oferecer um excelente ambiente de trabalho.

Agradeço à CAPES, ao Departamento de Informática da Universidade Federal de Viçosa e ao Departamento de Ciência da Computação da Universidade Federal de Minas Gerais, cujo apoio foi fundamental para a condução da pesquisa.

Agradeço à Professora Mariza Andrade da Silva Bigonha, cujas sugestões enriqueceram o trabalho de pesquisa. Agradeço também aos demais membros da banca examinadora, Professores Carlos Camarão de Figueiredo, Hermano Perrelli de Moura, José Lucas Mourão Rangel Netto e Roberto Ierusalimschy. Ao Professor Neil Jones, agradeço a atenção dedicada no período que visitei a Universidade de Copenhagen.

Aos colegas do Laboratório de Linguagens de Programação, agradeço o apoio, amizade e a agradável convivência: Fabio Tirelo, Elaine Gouveia Pimentel, Marcelo de Almeida Maia, Marcelo Luiz Silva, Marco Túlio Oliveira Valente, Patricia Costa, Marco Rodrigo Costa, Marcos Gonçalves Rios, Flávia Peligrinelli Ribeiro, Fernando Magno Quintão Pereira. Agradeço também a amizade de Genicio Zanetti e Carlos José Gomes da Rocha.

Agradeço especialmente a Anderson Luiz Caçador, que foi quem mais suportou meu mau humor, depressão e brincadeiras durante os quatro anos do curso.

Finalmente, agradeço o apoio dado por meus pais, Albino José Di Iorio e Maria José Oliveira Di Iorio, que sempre me incentivaram a estudar e a me tornar uma pessoa melhor.

Resumo

Avaliação Parcial é uma técnica de especialização de programas. Se um programa tem duas ou mais entradas, o resultado da avaliação parcial do mesmo em relação à primeira entrada é um novo programa, designado *residual* ou *especializado*. Quando o programa residual é executado sobre o restante das entradas, produz a mesma saída que o programa original, se executado sobre a entrada completa. O objetivo principal da avaliação parcial é o ganho em eficiência. Se parte da entrada é conhecida, toda informação que depende apenas dela pode ser previamente computada, podendo gerar de forma totalmente automática um programa especializado mais eficiente que o original.

Uma aplicação importante de avaliação parcial é a geração automática de compiladores dirigida por semântica. A partir de um interpretador de uma linguagem de programação, é possível produzir automaticamente um compilador. As técnicas de avaliação parcial podem ser usadas ainda na construção de um gerador de compiladores.

As *Máquinas de Estado Abstratas* (ASM, do inglês *Abstract State Machines*) são um formalismo criado por Yuri Gurevich, anteriormente conhecido por *Evolving Algebras*. É utilizado na descrição da semântica de linguagens de programação, arquiteturas de sistemas, protocolos distribuídos etc. Usando esse modelo, uma forma de se especificar a semântica de uma linguagem de programação é escrever um interpretador para essa linguagem. O interpretador recebe um programa da linguagem e seus dados de entrada, e simula a execução do programa sobre esses dados.

Neste trabalho, aplicamos as técnicas de avaliação parcial ao formalismo das Máquinas de Estado Abstratas. Como a maioria dos modelos para descrição de semântica, ASM sofre com implementações ineficientes. O nosso objetivo é produzir automaticamente especificações ASM especializadas e obter um ganho em eficiência que torne o uso desse modelo mais atraente. Como uma das principais aplicações de ASM é a descrição da semântica de linguagens de programação, a maioria dos experimentos conduzidos está relacionada à geração de compiladores dirigida por semântica.

O trabalho pode ser dividido em duas fases principais. Na primeira fase, desenvolvemos técnicas para a construção de um avaliador parcial para a linguagem das Máquinas de Estado Abstratas. Produzimos uma especificação desse avaliador parcial usando o próprio modelo ASM e demonstramos importantes propriedades sobre seu funcionamento, mostrando a facilidade que esse formalismo oferece para construção de especificações e demonstrações. Encontramos dificuldades, que consideramos inerentes ao modelo ASM, para a produção de um gerador de compiladores por meio da auto-aplicação do avaliador parcial.

Na segunda fase do trabalho, utilizamos uma técnica conhecida como *extensões de geração*. Essa abordagem consiste em construir à mão um gerador de compiladores usando técnicas de avaliação parcial, evitando os problemas associados à auto-aplicação de um avaliador parcial tradicional. Conduzimos experimentos com geração automática de compiladores, usando descrições da semântica de linguagens escritas em ASM.

Abstract

Partial Evaluation is a technique for program specialization. Let P be a program with two or more input data. Partially evaluating P with respect to the first input data generates a *residual* or *specialized* program. Running the residual program over the remaining input data will yield the same result as running P over all input data. The main goal of partial evaluation is to generate efficient programs from general ones by completely automatic methods.

Semantics based compiler generation is an important application of partial evaluation. A compiler can be automatically produced from an interpreter definition. The techniques can also be used to build a compiler generator.

Abstract State Machines are a formalism created by Yuri Gurevich, previously known as *Evolving Algebras*. It has been successfully used to describe the semantics of programming languages, architectures, distributed protocols etc. In ASM, the semantics of a programming language is usually given by an interpreter for the language. The interpreter simulates the execution of a program over its input data.

Our work involves partial evaluation and Abstract State Machines. A typical problem of most formalisms to describe semantics of programs, including ASM, is an inefficient implementation. Our goal is to automatically generate specialized ASM specifications, that are more efficient than the original ones. We hope that this will make ASM available even for applications which require more efficient implementations. The specification of the semantics of programming languages is an important application of ASM. So most experiments developed in our work are related to semantics based compiler generation.

Our work can be divided in two separate phases. In the first phase, techniques for building a partial evaluator for ASM have been developed. We have written a specification of this partial evaluator using the ASM language itself. We have proven important properties related to the partial evaluator, showing that ASM is suitable to describe algorithms in a clear way and to prove properties of these algorithms. We have had difficulties to automatically produce a compiler generator by self-application of the partial evaluator. We consider that these difficulties are inherent to the ASM model.

In the second phase of the work, we have used the *generating extension* approach. This approach consists in hand writing a compiler generator using partial evaluation techniques, avoiding the problems related to self-application of a traditional partial evaluator. We have carried out experiments involving automatic compiler generation, using descriptions of the semantics of programming languages written in ASM.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O trabalho desenvolvido envolve os conceitos de *Máquinas de Estado Abstratas* e *Avaliação Parcial de Programas*. Este capítulo apresenta uma definição sucinta desses conceitos, juntamente com uma rápida introdução à geração automática de compiladores, que é uma das principais aplicações de avaliação parcial de programas. Os objetivos e resultados alcançados pela pesquisa são também discutidos neste capítulo. Abordagens detalhadas de *Máquinas de Estado Abstratas* e *Avaliação Parcial de Programas* podem ser encontradas nos capítulos 2 e 3, respectivamente.

1.1 Máquinas de Estado Abstratas

As Máquinas de Estado Abstratas (ASM, do inglês *Abstract State Machines*) são um formalismo criado por Yuri Gurevich [40], originalmente conhecido por *Evolving Algebras*. É utilizado na descrição de arquiteturas de sistemas [15, 16, 19], linguagens de programação [20, 21, 22, 42, 84], sistemas distribuídos [7, 8, 17, 44] e de tempo real [35, 45], entre outros [18].

Usando esse formalismo, uma forma de se especificar a semântica de uma linguagem de programação L é escrever um interpretador para programas escritos em L. O interpretador recebe um programa S, escrito em L, e seus dados de entrada x, simulando a execução de S sobre x.

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. Assim, todo o trabalho se concentra em um subconjunto do modelo designado *ASM Seqüencial* [40].

1.2 Avaliação Parcial de Programas

Dado um programa P com entradas in_1 e in_2, a *avaliação parcial* de P em relação a in_1 é um novo programa P_{in_1} que recebe apenas uma entrada in_2 e produz os mesmos

resultados que P, se lhe fossem fornecidos in_1 e in_2 [55]. Avaliação parcial é um tipo de *especialização de programas*. O programa P_{in_1} é designado programa *especializado* ou *residual*. As entradas in_1 e in_2 são designadas, respectivamente, *estática* e *dinâmica*.

Um *avaliador parcial* para uma linguagem L é um programa que executa avaliação parcial sobre programas escritos em L, produzindo como resultado programas especializados que são escritos, geralmente, na mesma linguagem L. 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` [32].

Suponha disponível um avaliador parcial `mix` para uma linguagem L'. Suponha também que Int seja um interpretador de uma linguagem L, escrito em L', 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', que recebe x e produz a mesma saída de S. Ou seja, S' é o resultado da compilação de S para a linguagem L'. A avaliação parcial do próprio `mix`, dado como entrada estática um interpretador Int de L, resulta em um compilador de L para L'. Finalmente, a avaliação parcial de `mix` dado como entrada estática o próprio `mix` resulta em um gerador de compiladores, comumente chamado de `cogen`. Na realidade, `cogen` é um programa mais geral do que um gerador de compiladores. Se for aplicado a um interpretador, `cogen` produz um compilador. A aplicação de `cogen` a um programa P qualquer produz o que se convencionou chamar de *extensão de geração* (*g*enerating extension) para P [70].

A possibilidade da realização dos procedimentos descritos acima foi descoberta por Futamura [33] no início da década de 70, mas só foram realizados com sucesso pela primeira vez em meados dos anos 80 [57]. As equações que descrevem esses processos passaram a ser conhecidas como as *Três Projeções de Futamura*. A primeira projeção está relacionada à compilação por meio de avaliação parcial, a segunda projeção está relacionada à geração de compiladores e a terceira projeção de Futamura demonstra como um gerador de compiladores pode ser construído usando um avaliador parcial.

Uma alternativa para avaliação parcial é a abordagem de *extensões de geração* [70]. Essa abordagem consiste em escrever à mão o programa `cogen`, que pode ser gerado automaticamente por meio da auto-aplicação de um avaliador parcial. O programa `cogen` é denominado então *gerador de extensões de geração*. Se aplicado a um programa P com duas entradas, é produzido um novo programa P_{gen}, designado *extensão de geração* para P. Se for fornecida a P_{gen} a primeira entrada in_1 de P, é produzido um programa P_{in_1}, que é a especialização de P em relação à sua primeira entrada. Em suma, um gerador de extensões de geração produz avaliadores parciais específicos para um determinado programa. A construção de um gerador de extensões de geração é geralmente mais complicada que a construção de um avaliador parcial tradicional, se não forem considerados problemas relacionados à auto-aplicação. Por outro lado, um compilador é gerado automaticamente pela aplicação de um gerador de extensões de geração a um interpretador, sem a necessidade da auto-aplicação de um avaliador parcial, como discutido acima.

Avaliadores parciais foram construídos com sucesso para linguagens de paradigma imperativo [2, 3, 66], funcional [54, 60, 11, 68] e lógico [46, 78, 69]. Embora as técnicas utilizadas sejam semelhantes, cada paradigma e até mesmo cada linguagem possui características especiais que necessitam de adaptações específicas. Cada construção especial de

uma linguagem pode demandar a adaptação ou criação de novas técnicas de avaliação parcial. O mesmo é válido para a abordagem de extensões de geração. O uso dessa abordagem é mais recente, sendo aplicada com sucesso principalmente em linguagens estaticamente tipadas [50, 12, 13, 80].

1.3 Geradores de Compiladores

Ferramentas como geradores de analisadores sintáticos e avaliadores de gramáticas de atributos são muito utilizadas para geração de compiladores. Nessas ferramentas, o usuário fica totalmente responsável por garantir a consistência do código gerado quanto à semântica da linguagem compilada. Geradores automáticos dirigidos por semântica evitam esse problema. Se for estabelecida uma prova formal da correção do gerador, os compiladores produzidos serão também automaticamente corretos em relação à semântica da linguagem descrita.

Os primeiros geradores de compiladores dirigidos por semântica se baseavam em semântica denotacional, como o de Mosses [71], Jones e Schmidt [56], os sistemas CERES [83] e *Script* [10]. O modelo, embora bastante elegante, levava à obtenção de compiladores extremamente ineficientes, mesmo com testes simples. Alguns trabalhos, ainda adotando a semântica denotacional, procuraram melhorar a eficiência dos compiladores gerados [38, 76, 64, 65]. Resultados expressivos foram conseguidos usando-se semântica de ações, como em [75].

A compilação e a geração de compiladores usando avaliação parcial foram realizadas com sucesso em diversos experimentos. A maioria dos experimentos de sucesso usava uma linguagem funcional de avaliação estrita como linguagem objeto. Kahn e Carlsson realizaram a compilação de programas Prolog para Lisp, por meio da avaliação parcial de um interpretador de Prolog escrito em Lisp [62]. Consel e Khoo geraram um compilador de Prolog para Scheme, por meio da aplicação da Segunda Projeção de Futamura [24]. Jorgensen gerou um compilador de um subconjunto de Miranda para Scheme com eficiência comparável a compiladores escritos à mão [60, 61].

1.4 Objetivos e Resultados Alcançados

Acreditamos que as Máquinas de Estado Abstratas constituam um formalismo adequado para se descrever a semântica de linguagens de programação e de outros sistemas. O objetivo principal do nosso trabalho de pesquisa foi o desenvolvimento de técnicas de avaliação parcial adequadas ao modelo ASM, possibilitando a geração automática de especificações mais eficientes.

Quando as especificações são interpretadores de linguagens de programação, tem-se descrições da semântica dessas linguagens. Usando técnicas de avaliação parcial, pode-se produzir automaticamente compiladores para essas linguagens, ou seja, geração de compiladores dirigida por semântica. Como uma das principais aplicações de ASM é a descrição da semântica de linguagens de programação, o trabalho teve uma ênfase maior na aplicação

de avaliação parcial para geração automática de compiladores.

Foram desenvolvidas técnicas e ferramentas para aplicar a teoria de avaliação parcial sobre a linguagem das Máquinas de Estado Abstratas. Entre as características especiais da linguagem, que demandam adaptações das técnicas de avaliação parcial, estão a execução paralela de comandos e a atualização de funções.

O trabalho desenvolvido pode ser dividido em duas partes principais. A primeira parte compreendeu o desenvolvimento de técnicas para implementar um avaliador parcial `mix` para a linguagem das Máquinas de Estado Abstratas. Problemas com a auto-aplicação foram encontrados. Foram demonstradas formalmente propriedades importantes do avaliador parcial e dos programas por ele produzidos. Na segunda parte do trabalho, os esforços foram concentrados no desenvolvimento de um gerador de extensões de geração para ASM. Nessa segunda etapa, resultados práticos na geração de compiladores foram obtidos com mais facilidade.

1.4.1 Avaliador Parcial para ASM

Nesta seção, vamos apresentar uma introdução à primeira parte do trabalho, cujo resultado principal foi o desenvolvimento de um avaliador parcial `mix` para a linguagem das Máquinas de Estado Abstratas. Uma abordagem mais detalhada pode ser encontrada nos capítulos 4 e 5.

Qualquer linguagem de programação pode ser utilizada para implementar o avaliador parcial `mix` para ASM. No nosso caso, a implementação foi conduzida em Java. O avaliador parcial recebe como entrada uma especificação escrita em ASM e parte da sua entrada, produzindo uma especificação especializada.

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*.

Foi também implementada uma versão de `mix` escrita na própria linguagem ASM. Isso nos permite a auto-aplicação, ou seja, avaliar parcialmente o próprio `mix`. Se o interpretador de uma linguagem L for fornecido como entrada estática, 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 por Huggins e Gurevich em [43], mas não permitia a auto-aplicação e conseqüentemente não possibilitava a geração de compiladores.

Usando a versão do avaliador parcial para ASM implementada em Java, foram conduzidos testes que envolveram sua aplicação sobre algumas descrições de semântica de linguagens, como por exemplo, um subconjunto de C. O resultado, seguindo a Primeira Projeção de Futamura, foi a compilação de programas escritos nessas linguagens para a linguagem das Máquinas de Estado Abstratas. Nessa versão, a linguagem dos programas ASM fornecidos como entrada é não tipada.

Como explicado anteriormente, foi também desenvolvida uma segunda versão do avaliador parcial, escrita na própria linguagem ASM dinamicamente tipada. Esse avaliador,

que foi designado $\texttt{mix}_{\text{ASM}}$, pode servir de entrada para o primeiro avaliador parcial escrito em Java. Essa "auto-aplicação", de acordo com a Segunda Projeção de Futamura, produz um compilador a partir da especialização de um avaliador parcial em relação a um interpretador. Entretanto, $\texttt{mix}_{\text{ASM}}$ carece de várias otimizações, o que tornou a auto-aplicação restrita. Foram realizados testes que consistiram na especialização de $\texttt{mix}_{\text{ASM}}$ em relação a interpretadores simples, como, por exemplo, um interpretador para a Máquina de Turing. O resultado alcançado, como esperado, foi a geração de compiladores das linguagens descritas para a linguagem das Máquinas de Estado Abstratas. Os compiladores gerados são, entretanto, bastante ineficientes.

Devido a dificuldades em se obter um compilador eficiente usando auto-aplicação, desisitimos de ir além e aplicar a Terceira Projeção de Futamura. O objetivo seria a geração de um gerador de compiladores para ASM. Um problema sério que contribuiu para essa decisão é discutido na Seção 4.5. Sendo assim, adotamos uma nova abordagem que também utiliza técnicas de avaliação parcial, denominada abordagem de extensões de geração, que discutiremos a seguir.

Os resultados descritos nesta seção foram publicados inicialmente em [28]. Nesse artigo, as técnicas utilizadas para implementar um avaliador parcial para ASM foram apresentadas. Mais tarde, um novo artigo foi produzido, descrevendo os resultados da geração de compiladores simples com a auto-aplicação do avaliador parcial [30].

1.4.2 Extensões de Geração para ASM

Os trabalhos envolvendo avaliação parcial e ASM fazem parte de uma série de projetos do grupo de pesquisa em Linguagens de Programação do DCC-UFMG, relacionados à utilização das Máquinas de Estado Abstratas. Um dos projetos é a proposta de uma versão estaticamente tipada para a linguagem das Máquinas de Estado Abstratas, designada *Machĭna* [81]. A linguagem Machĭna foi projetada de modo a facilitar uma compilação eficiente para linguagens imperativas comuns.

A geração de compiladores por meio da aplicação das projeções de Futamura não é muito adequada a linguagens estaticamente tipadas. A abordagem indicada, nesse caso, é a construção de um gerador de extensões de geração. Como discutido na Seção 1.2, essa abordagem consiste em escrever à mão o programa `cogen`, que pode ser gerado automaticamente por auto-aplicação de um avaliador parcial.

Uma das razões de termos adotado inicialmente a abordagem das projeções de Futamura foi devido ao fato de não termos disponível uma versão estaticamente tipada da linguagem ASM quando o trabalho foi iniciado. A proposta da linguagem Machĭna surgiu quando boa parte da pesquisa com avaliação parcial já havia sido conduzida. Além disso, descobrimos que algumas construções da linguagem ASM tornavam muito difícil obter bons resultados com auto-aplicação. Esses motivos fizeram com que a construção de um gerador de extensões de geração passasse a fazer parte dos nossos objetivos.

Entretanto, um compilador confiável para Machĭna não foi implementado a tempo. O nosso procedimento, então, foi desenvolver um gerador de extensões de geração cujo núcleo processa construções básicas de uma linguagem abstrata baseada no modelo das Máquinas de Estado Abstratas. Para o funcionamento do sistema, devem ser acoplados módulos

de leitura e geração para uma linguagem concreta específica. Nos testes que realizamos, utilizamos a linguagem Xasm, que possui um compilador para C bastante utilizado [6]. Um módulo para a linguagem Machĭna poderá ser desenvolvido e testado no futuro, quando o compilador estiver disponível.

Usando o gerador de extensões de geração e o compilador de Xasm, foram conduzidos testes de geração de compiladores para linguagens mais complexas. Por exemplo, a partir da descrição da semântica de um significativo subconjunto da linguagem C, escrito em Xasm, geramos automaticamente um compilador dessa linguagem para Xasm. Os detalhes de implementação e os resultados dos testes são descritos no Capítulo 7.

As contribuições da nossa pesquisa podem ser comparadas principalmente com os resultados dos trabalhos envolvendo avaliação parcial e geração de compiladores, como em [62, 24, 60, 61]. De maneira diversa desses trabalhos, procuramos implementar um ambiente onde a linguagem objeto é uma linguagem imperativa eficiente (linguagem C), sem abrir mão de uma linguagem de descrição de semântica de alto nível (modelo ASM).

1.5 Organização do Documento

O Capítulo 2 apresenta o formalismo das Máquinas de Estado Abstratas. O texto concentra-se nos aspectos relativos ao uso do modelo para descrição da semântica de linguagens de programação, apresentando o subconjunto de ASM utilizado para especificar sistemas seqüenciais. A Seção 2.1 apresenta o modelo de maneira informal. As ASMs são caracterizadas como máquinas abstratas cujos estados são representados por funções e relações. Um programa para a máquina é uma regra de transição, aplicada sobre o estado corrente para produzir o próximo estado. Entre as principais regras de transição, são apresentadas a regra de atualização, a regra condicional e o construtor de blocos. Uma série de exemplos pode ser encontrada na Seção 2.2, tornando mais claro o entendimento dos conceitos envolvidos. A Seção 2.3 repete a maioria dos conceitos, desta feita aplicando uma abordagem mais formal. Um leitor familiarizado com o modelo ASM pode dispensar a leitura desse capítulo. Para entender os conceitos discutidos nos capítulos seguintes, pode ser suficiente a leitura das seções 2.1 e 2.2, além das conclusões.

O Capítulo 3 discute aspectos teóricos e práticos relacionados à avaliação parcial de programas. Apresentamos uma descrição bastante completa, cobrindo diferentes abordagens, combinando aspectos teóricos e práticos e fornecendo muitos exemplos. Uma versão ainda mais desenvolvida desse texto foi apresentada como um tutorial sobre avaliação parcial de programas [29]. Se o leitor conhecer o assunto, pode dispensar a leitura do texto. Para entender os conceitos abordados neste documento, recomenda-se pelo menos a leitura da introdução do capítulo (Seção 3.1) e da Seção 3.5. Essa última seção traz uma abordagem mais formal do assunto e discute detalhadamente a aplicação de avaliação parcial na geração de compiladores, explorando a abordagem das projeções de Futamura e a abordagem de extensões de geração. Recomenda-se ler também as conclusões.

As técnicas utilizadas para desenvolver um avaliador parcial para ASM são discutidas no Capítulo 4. Esse capítulo apresenta resultados alcançados com a utilização das Projeções de Futamura para compilação e geração de compiladores.

Os detalhes de uma implementação do avaliador parcial para ASM na própria linguagem ASM são apresentados no Capítulo 5. Propriedades importantes sobre o funcionamento desse avaliador parcial, bem como dos programas gerados, são formalmente demonstradas no Capítulo 6.

O Capítulo 7 apresenta o gerador de extensões de geração para ASM. A geração de compiladores para linguagens mais complexas, como um significativo subconjunto da linguagem C, é o resultado mais expressivo desse capítulo.

As conclusões finais são apresentadas no Capítulo 8.

Capítulo 2

Máquinas de Estado Abstratas

Este capítulo apresenta o modelo de especificação formal ASM, anteriormente conhecido como *Evolving Algebras*. O texto utilizado foi, em grande parte, extraído do tutorial de ASM apresentado em [82].

Máquinas de Estado Abstratas (ASM, do inglês *Abstract State Machines*), introduzidas por Yuri Gurevich em [39, 40], constituem um conceito expressivo e elegante para modelagem matemática de sistemas dinâmicos discretos.

A idéia original era descrever a semântica operacional para algoritmos elaborando a tese implícita de Turing, isto é, "todo algoritmo pode simulado por uma máquina de Turing apropriada". Desta maneira, Turing descreveu a semântica operacional de algoritmos. Entretanto, esta semântica é muito inconveniente, pois pode ser necessária uma longa seqüência de passos na máquina de Turing para simular um único passo do algoritmo. Assim, as ASM foram criadas com o objetivo de simular algoritmos de maneira mais natural [39]. A Tese de ASM Seqüencial diz que "todo algoritmo seqüencial, em qualquer nível de abstração, pode ser visto como uma ASM" [41].

As ASM possuem recursos para modelar interação do programa com o "mundo exterior", possibilitando formalizar as ações do ambiente no qual o sistema está inserido. A metodologia de ASM provê recursos expressivos para especificar a semântica operacional de sistemas dinâmicos discretos, em um nível de abstração natural e de uma maneira direta e essencialmente livre de codificação [27]. Com isso, tem-se por objetivo diminuir a distância que há entre modelos formais de computação e métodos práticos de especificação. A metodologia utiliza conceitos simples e bem conhecidos, o que facilita a leitura e a escrita de especificações de sistemas.

Na literatura, há vários exemplos de utilização de ASM na especificação formal de sistemas, dentre os quais podemos citar: arquiteturas [15, 16, 19], linguagens de programação [20, 21, 22, 42, 84], sistemas distribuídos [7, 8, 17, 44] e de tempo real [35, 45], entre outros [18].

Interessa-nos especialmente o uso de ASM para especificar semântica de linguagens de programação. A semântica de uma linguagem L é geralmente dada por um interpretador escrito em ASM, que recebe como entradas um programa P escrito em L e a entrada desse programa. O interpretador simula a execução de P sobre sua entrada. Vamos nos concentrar nas linguagens de programação seqüenciais, isto é, as que não permitem

execução paralela ou concorrente de processos.

Neste capítulo, vamos apresentar o conceito de Máquinas de Estado Abstratas Seqüenciais, que são as ASMs usadas para especificar sistemas seqüenciais. Embora contenham primitivas para disparo de regras em paralelo, como veremos adiante, não permitem execução de vários agentes simultaneamente. ASM possibilita especificação de modelos multiagentes de modo simples e elegante [40], mas vamos nos concentrar apenas no modelo seqüencial. Na Seção 2.1, os conceitos são introduzidos de uma maneira informal. Na Seção 2.2, vários exemplos de uso são apresentados. A Seção 2.3 contém a definição formal do modelo. As conclusões deste capítulo são apresentadas na Seção 2.4.

2.1 Introdução ao Modelo ASM

Esta seção apresenta o modelo ASM de maneira informal.

Em linhas gerais, as ASMs são máquinas abstratas, cujos estados são formados por funções e relações. Estas funções ou relações possuem nomes e são definidas em um conjunto denominado o *superuniverso* do estado. O conjunto dos nomes de funções e relações de um estado é o *vocabulário* do estado. A interpretação de um nome de função ou relação é um mapeamento dos nomes pertencentes ao vocabulário nas respectivas funções ou relações.

A mudança de estado da máquina é dada por uma regra de transição. Uma regra de transição modifica a interpretação de alguns nomes de função do vocabulário do estado. Uma regra de transição de ASM tem a forma de um programa como de uma linguagem imperativa comum. A diferença principal é a ausência de iteração, pois esse conceito está implícito na execução da máquina. Com efeito, a execução da máquina consiste em executar a sua regra de transição repetidas vezes, modificando a cada vez o estado atual. Dessa forma, é formada uma seqüência de estados de mesmo vocabulário e compostos por diferentes funções e relações. Dizemos que a interpretação dos nomes pertencentes ao vocabulário é modificada de estado para estado durante a execução.

Mais precisamente, um *vocabulário* Υ é um conjunto de nomes de funções e relações, cada nome com uma aridade fixa associada. Por exemplo, suponha que a aridade da função de nome i seja 0 e da função de nome f seja 1. Temos que o conjunto $\{i, f\}$ é um vocabulário. Os nomes das relações de zero argumento *true*, *false*, o nome da função de zero argumento *undef*, os operadores booleanos usuais e o sinal de igualdade estão presentes em todo vocabulário.

Um *estado* S de vocabulário Υ é um conjunto X, denominado o *superuniverso* de S, junto com as interpretações, em X, dos nomes de funções e relações pertencentes a Υ.

Se f é um nome de função de aridade r, então f é interpretado como uma função $\boldsymbol{f} : X^r \rightarrow X$. Se f é um nome de relação de aridade r, então f é interpretado como uma função $\boldsymbol{f} : X^r \rightarrow \{true, false\}$. Se U é um nome de relação pertencente a Υ, então o conjunto $U = \{\bar{x} : U(\bar{x}) = true\}$ é um *universo* contido em X. Neste caso, dizemos que $U(\bar{x}) = true$ e $\bar{x} \in U$ são equivalentes.

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

Uma regra de atualização é da forma

$$R \equiv f(\bar{x}) := y,$$

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

Uma regra condicional da forma

$$R \equiv \textbf{if } g \textbf{ then } R_1 \textbf{ else } R_2 \textbf{ endif}$$

tem a seguinte semântica: se a expressão g avaliar em *verdadeiro*, então o estado resultante é o resultado da regra R_1; caso contrário, o estado resultante é o resultado da regra R_2.

Uma regra bloco da forma

$$R \equiv R_1, \ldots, R_n$$

tem a seguinte semântica: o estado formado por R é o resultado da execução de todas as regras R_i em paralelo. Por exemplo, a execução da regra bloco

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

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

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

Example 1 *(Função Fatorial)*

Suponha que o estado inicial S_0 seja dado com o vocabulário $\Upsilon = \{f, i\}$, onde i é um nome de função de zero argumento, interpretado como 0*, e f é um nome de função unária, interpretado como a função $\boldsymbol{f} = \lambda x.(x = 0 \to 1, \mathit{undef})$[1].A regra da especificação é o bloco:*

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

Ao executarmos a regra no estado S_0, obtemos um novo estado S_1, no qual i é interpretado como 1 *e f é interpretado como a função*

$$\boldsymbol{f} = \lambda x.(x = 0 \to 1, (x = 1 \to 1, \mathit{undef}))$$

Como podemos ver na Figura 2.1, ao executarmos a regra no estado S_1, obtemos um novo estado S_2, no qual i é interpretado como 2 e a interpretação de f, no ponto 2, passa a ser igual a 2. Da mesma forma, obtemos os estados $S_3, S_4, \ldots$.

Pode-se ver facilmente que, no estado S_k, $k \geq i$, a interpretação de f, $\boldsymbol{f}$, é uma função que, aplicada a $\boldsymbol{i}$, retorna $\boldsymbol{i}!$.

[1] *Se $f = \lambda x.E$, então f é uma função dada por $f(x) = E$. Mais detalhes sobre a notação lambda podem ser encontrados em [59]. A notação $a \to b, c$, utilizada em [37], é equivalente a* ***if*** *a* ***then*** *b* ***else*** *c.*

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

Figura 2.1: Interpretação do nome de função *f*.

Um aspecto importante que deve ser modelado na especificação de sistemas é que, em geral, eles são afetados pelo ambiente. O ambiente se manifesta por meio de algumas funções básicas, chamadas *funções externas*. Um exemplo típico de função externa é uma entrada fornecida pelo usuário. Pode-se pensar em funções externas como *oráculos*, tais que, a especificação fornece argumentos e o oráculo fornece o resultado [40].

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

Uma regra *import* é da forma

$$\textbf{import } v \ R_0 \textbf{ endimport},$$

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

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

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

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

$$\textbf{extend } U \textbf{ with } v \ R_0 \textbf{ endextend}.$$

Uma regra *choose* é da forma

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

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

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

Uma regra *var* é da forma

$$\textbf{var } v \textbf{ ranges over } U \ R_0 \textbf{ endvar},$$

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

Para simplificar as especificações, vamos supor que conjuntos como o dos números inteiros, dos racionais, das listas e dos conjuntos estejam contidos no superuniverso de um estado.

2.2 Exemplos

Nesta seção, apresentaremos alguns exemplos de definição de sistemas em ASM.

Example 2 *Pesquisa Binária*

Este exemplo tem por finalidade mostrar a utilização de funções e regras básicas para especificação de algoritmos simples.

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

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

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

A regra da especificação é

if ($\neg$ *encontrado* ***and*** $inf \leq sup$) ***then***
  if ($k = f((inf + sup)/2)$) ***then***
    $encontrado := true$,
    $pos := (inf + sup)/2$
  elseif ($k < f((inf + sup)/2)$) ***then***
    $sup := (inf + sup)/2 - 1$
  else
    $inf := (inf + sup)/2 + 1$
  endif
endif

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

Example 3 *Ordenação por seleção*

Da mesma forma que no Exemplo 2, a função f representa os elementos que desejamos ordenar. Quando a execução convergir, esperamos que se $1 \leq i \leq j \leq n$, então $f(i) \leq f(j)$.

No estado inicial, o nome de função Modo é interpretado como 1, i é interpretado como 1 e f é interpretado como no Exemplo 2. Todos os outros nomes de função são interpretados como undef.

A regra da especificação é a seguinte:

if $Modo = 1$ ***and*** $i < n$ ***then***
    $k := i$**,** $j := i + 1$**,** $Modo := 2$
elseif $Modo = 2$ ***then***
    if $j > n$ ***then***
        $Modo := 3$
    elseif $f(j) < f(k)$ ***then***
        $k := j$**,** $j := j + 1$
    endif
elseif $Modo = 3$ ***then***
    if $k \neq i$ ***then***
        $f(k) := f(i)$**,** $f(i) := f(k)$
    endif
    $i := i + 1$**,** $Modo := 1$
endif

A máquina executa até que i atinja o valor n. Neste estado, o arranjo está ordenado. A função Modo é utilizada para que o modelo, essencialmente paralelo, simule a execução dos passos seqüenciais do algoritmo. Observe que a permutação de dois valores do arranjo não demanda a utilização de um temporário.

Example 4 *Números Primos*

Neste exemplo, especificaremos um algoritmo bastante simples para encontrar todos os primos menores ou iguais a um número n qualquer. Para isso, definiremos um universo Numeros, subconjunto dos números inteiros, tal que $Numeros = \{x \in Inteiros : 2 \leq x \leq n\}$.

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

A regra da especificação é a seguinte:

if $x \leq n$ ***then***
    var y ***ranges over*** $Numeros$
        if $y < x$ ***and*** $x\%y = 0$ ***then***
            $primo(x) := false$
        endif
    endvar,
    $x := x + 1$
endif

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

Example 5 *Interpretador para a Máquina de Turing*

Este exemplo exibe um interpretador de uma versão da Máquina de Turing. *Um interpretador para uma linguagem escrito em ASM pode ser visto como uma maneira geral de se especificar a semântica dessa linguagem. Assim, este exemplo especifica a semântica da linguagem de uma versão da Máquina de Turing, descrita a seguir.*

A máquina possui as seguintes instruções: `right`, `left`, `write a`, `goto i`, `if a goto i`. *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`*. O programa exemplo apresentado a seguir, extraído de [55], 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
```

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 possa 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`*. No estado inicial,* `cont` *indica a primeira instrução do programa MT e* `cabeca` *indica a primeira célula da fita com valor diferente de* `undef`*, se houver. O código ASM é exibido abaixo:*

if $cod(cont) = LEFT$ ***then***
  $cont := cont + 1$**,**
  $cabeca := cabeca - 1$
elseif $cod(cont) = RIGHT$ ***then***
  $cont := cont + 1$**,**
  $cabeca := cabeca + 1$
elseif $cod(cont) = WRITE$ ***then***

$cont := cont + 1$,
$fita(cabeca) := par1(cont)$
elseif $cod(cont) = GOTO$ ***then***
$cont := par1(cont)$
elseif $cod(cont) = IFGOTO$ ***then***
if $par1(cont) = fita(cabeca)$ ***then***
$cont := par2(cont)$
else
$cont := cont + 1$
endif
endif

2.3 O Modelo Formal de ASM

Nesta seção apresentaremos o modelo formal de ASM.

2.3.1 Definição da Máquina

ASM são sistemas de transição que especificam computações cujos estados são álgebras [27]. Os universos de álgebras que formam os estados da computação constituem o superuniverso da ASM.

Definition 1 *(Vocabulário) Um* vocabulário Υ *é uma coleção finita de nomes de função ou relação, cada nome com uma aridade fixa. Estão presentes em todo vocabulário: o sinal de igualdade, true, false e undef e os operadores booleanos usuais.*

A função *undef* é utilizada para modelar funções parciais.

Definition 2 *(Estado) Um* estado S *é uma álgebra [34], dada por:*

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

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

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

Definition 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, dadas abaixo. Basicamente, o caráter dinâmico do sistema é modelado pelas alterações, de estado para estado, na interpretação de funções dinâmicas.

Definition 4 *(Termo)* Termos *são definidos recursivamente como:*

- *uma variável é um termo;*
- *um nome de função ou relação de zero argumento é um termo;*
- *se f é um nome de função ou relação r-ária, $r > 0$, e $\bar{x} = (x_1, \ldots, x_r)$ é uma tupla onde cada $x_i, 1 \leq i \leq r$ é um termo, então $f(\bar{x})$ é um termo.*

Termos sem variáveis são interpretados, em um estado S, da maneira descrita a seguir:

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

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

A interpretação de termos que contêm variáveis será mostrada na Seção 2.3.2, onde definiremos as regras não-básicas, isto é, as regras que utilizam variáveis.

Definition 5 *(Endereço) Um* endereço *em um estado $S = (\Upsilon, X, Val)$ é um par $(f, \bar{x})$, onde $f \in \Upsilon$ é um nome de função dinâmica, e $\bar{x} \in X^*$ é uma tupla cujo comprimento é igual à aridade de f.*

Definition 6 *(Atualização) Uma* atualização *em um estado $S = (\Upsilon, X, Val)$ é um par (l, y), onde l é um endereço em S e $y \in X$.*

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

Definition 7 *(Consistência de um Conjunto de Atualizações) O conjunto de atualizações Δ é* consistente*, se*

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

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

Definition 8 *(Regra)* Regras *são definidas recursivamente por:*

- *A regra de atualização* $R \equiv f(\bar{t}) := t_0$ *é uma regra. Nesta expressão,* f *é um nome de função dinâmica,* t_0 *é um termo e* $\bar{t}$ *é uma tupla de termos cujo comprimento é equivalente à aridade de* f*. Se* f *é um nome de relação, então* t_0 *deve ser booleano. Definiremos o conjunto de atualizações de* R *em um estado* S*,* $Updates(R, S)$*, como o conjunto* $\{(l, y)\}$*, onde* $l = (f, Val_S(\bar{t}))$ *e* $y = Val_S(t_0)$*.*

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

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

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

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

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

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

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

Estas regras são conhecidas como regras básicas*.*

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

$$R_1; R_2$$

onde primeiro executa-se R_1 e em seguida R_2. Isto porque um programa em ASM descreve somente um passo do algoritmo. A composição seqüencial pode gerar estruturas de execução muito complexas, o que pode tornar o raciocínio sobre a especificação muito mais difícil [41].

Definition 9 *(Disparo de uma Regra) O* disparo de uma regra R *em um estado* S *produz um novo estado* S'*, tal que, se* $Updates(R, S)$ *for consistente, então*

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

Se o conjunto $Updates(R, S)$ *não for consistente, então o efeito do disparo de* R *em* S *será nulo, isto é, o estado* S' *resultante será igual a* S*.*

Definiremos $Fire(R, S)$ *como o estado resultado do disparo de* R *em* S*.*

Outra abordagem utilizada para tratar conjuntos de atualização inconsistentes é fazer uma escolha não-determinista da atualização que será disparada [52].

Definition 10 *(Especificação ASM) Uma* especificação ASM *é uma tupla* $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$*, onde*

- $\Upsilon = \Upsilon_0 \cup \Upsilon_e$ *é um vocabulário composto pela união de um vocabulário pré-definido,* Υ_0*, e um vocabulário especificado pelo usuário,* Υ_e*. O vocabulário pré-definido* Υ_0 *contém os nomes de relação Boolean, Integer, String, List, Set, que serão interpretados, respectivamente, como os universos dos valores lógicos, dos números inteiros, das cadeias de caracteres, das listas e das partes de conjuntos.* Υ_0 *contém também as operações usuais sobre estes conjuntos;*

- $\mathcal{A}$ *é um conjunto de* Υ*-álgebras ou estados* S*, cada um consistindo no vocabulário* Υ *e um conjunto* X *(o superuniverso de* S*) de funções, que são as interpretações em* X *dos nomes de funções de zero argumento pertencentes a* Υ*. Um nome de função de aridade* r*,* $r > 0$*, é interpretado como uma função de* $X^r \to X$*. O conjunto* X *é comum a todos os estados pertencentes a* $\mathcal{A}$*;*

- $S_0 \in \mathcal{A}$ *é o estado inicial, onde as interpretações de alguns nomes de funções são dadas; nomes de funções cujas interpretações não são dadas em* S_0 *são interpretados como undef;*

- $\mathcal{P}$ *é uma regra que descreve as modificações das interpretações de nomes de funções de uma álgebra (estado) para outra.*

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

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

Definition 11 *(Execução de uma Especificação ASM) A* execução de uma especificação ASM $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$ *é uma seqüência* $\mathcal{S} = \langle S_n : n \geq 0 \rangle$ *de estados pertencentes a* $\mathcal{A}$*, onde* $S_{n+1} = Fire(\mathcal{P}, S_n)$*, isto é,* S_{n+1} *é obtido a partir de* S_n*, disparando a regra* $\mathcal{P}$ *em* S_n*.*

Na maioria dos sistemas dinâmicos discretos, a execução pode ser influenciada pelo ambiente. Intuitivamente, um ambiente ativo é um agente externo. Desta maneira, 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; na Seção 2.3.2 mostraremos uma construção para especificarmos explicitamente o não-determinismo.

2.3.2 Regras Não-Básicas

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

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

Definition 12 *(Regra* Import*) A regra* import *é uma regra da forma*

$$R \equiv \textbf{import } v \ R_0 \textbf{ endimport},$$

onde v é uma variável e R_0 é uma regra.

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

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

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

Definition 13 *(Regra* Var*) A regra* var *é uma regra da forma*

$$R \equiv \textbf{var } v \textbf{ ranges over } U \ R_0 \textbf{ endvar},$$

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

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

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

O efeito desta regra é criar uma instância de R_0 para cada elemento pertencente ao universo U. Em cada instância de R_0, a variável v está associada ao elemento correspondente de U. Após criadas as instâncias, todas são executadas em paralelo. Esta regra é usada para modelar paralelismo síncrono, como foi mostrado no Exemplo 4.

Definition 14 *(Regra* Choose*)*

A regra choose *é uma regra da forma*

$$\textbf{choose } v \textbf{ in } U \textbf{ satisfying } g \ R_0 \textbf{ endchoose},$$

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

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

2.4 Conclusões

Neste capítulo, apresentamos o modelo de especificação formal ASM. Os principais conceitos abordados foram:

- ASMs são máquinas abstratas cujos estados são representados por álgebras, ou seja, funções e relações.
- Uma execução de um programa é uma seqüência de estados, onde o próximo estado é produzido por meio da execução de uma regra de transição sobre o estado anterior.
- As principais regras de transição são a regra de atualização, a regra condicional e o construtor de blocos.
- A regra de atualização modifica o valor de uma função em um determinado ponto, de modo similar a um comando de atribuição em uma linguagem imperativa comum.
- A regra condicional determina uma guarda que impõe uma execução condicional de outras regras.
- O construtor de blocos reúne um conjuntos de regras, que serão executadas paralelamente.

Os programas em ASM são muito similares aos programas de linguagens imperativas comuns. Vimos que uma característica diferente é a forma de representar um estado, por meio de funções. Os programas não contêm um mecanismo de iteração explícito, pois a regra de transição é aplicada repetidamente sobre o estado corrente, até que um ponto fixo seja atingido. Outra característica que deve ser ressaltada é a execução paralela das regras. Vários exemplos de formalização usando o modelo foram exibidos na Seção 2.2.

O modelo ASM é bastante geral, assim propositalmente não especifica uma série de características necessárias a uma definição de uma linguagem de programação real, como por exemplo um sistema de tipos. Na nossa pesquisa, trabalhamos com duas implementações reais do modelo. A primeira implementação possui um interpretador escrito em Java e é

utilizada no Capítulo 4. A segunda é a linguagem Xasm [6], que possui um compilador para C e é utilizada no Capítulo 7.

O modelo ASM tem sido utilizado com sucesso para formalizar sistemas seqüenciais, paralelos e distribuídos. Neste capítulo, nos concentramos no subconjunto de ASM utilizado para formalizar sistemas seqüenciais, uma vez que o nosso principal interesse é sua utilização para descrever a semântica de linguagens de programação. A semântica de uma linguagem de programação é descrita em ASM por meio da especificação de um interpretador para a linguagem.

Como discutido no Capítulo 1, um dos principais objetivos de nosso trabalho é a geração automática de compiladores dirigida por semântica, usando o modelo ASM e avaliação parcial de programas. A partir de um interpretador escrito em ASM, um compilador é automaticamente gerado. O Capítulo 3 a seguir apresenta uma abordagem detalhada da teoria de avaliação parcial de programas, sem estar ligado a características específicas do modelo ASM. A utilização desse modelo está presente nos capítulos seguintes, onde são apresentados um avaliador parcial e um gerador de extensões de geração para ASM.

Capítulo 3

Avaliação Parcial

Este capítulo apresenta os principais conceitos relacionados à avaliação parcial de programas. O capítulo contém definições, exemplos, aplicações e tópicos de pesquisa.

As referências principais para a construção do texto foram o livro de Jones, Gomard e Sestoft [55], um relatório de Mogensen e Sestoft [70], e o material produzido por John Hatcliff para a Escola de Verão do DIKU de 1998 (*Partial Evaluation: Practice and Theory*). A Seção 3.6 apresenta essas e outras fontes de referênciade forma comentada.

A Seção 3.1 apresenta uma definição informal de avaliação parcial de programas e um exemplo simples. Resume os principais assuntos discutidos neste capítulo, exibindo sua distribuição nas demais seções. Pode servir como guia para a leitura do resto do capítulo.

3.1 Introdução

Suponha um programa P que tenha duas entradas, identificadas como in_1 e in_2. O resultado da *avaliação parcial* de P em relação à entrada in_1 é um novo programa P_{in_1}, designado por *residual* ou *especializado* [55]. O programa P_{in_1}, quando executado sobre a entrada restante in_2, produz o mesmo resultado que a execução de P sobre ambas as entradas. Avaliação parcial é um tipo de *especialização de programas*. A entrada in_1 é denominada *entrada estática*, e in_2, *entrada dinâmica*.

O objetivo principal da avaliação parcial é o ganho em eficiência. Se parte dos dados de entrada de um programa é conhecida, as estruturas do programa que dependam apenas dessa parte podem ser previamente computadas. O programa especializado conterá apenas o código necessário para processar os dados ainda não conhecidos.

Um *avaliador parcial* para uma linguagem L é um programa que realiza avaliação parcial de programas escritos em L, produzindo como resultado programas especializados que são escritos, geralmente, na mesma linguagem L. 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` [32, 58].

Observe a função *Power(n,x)* na Figura 3.1, escrita na linguagem C, que computa o valor de x^n. 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

```
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;
}
```

Figura 3.1: Função Power(n,x) = x^n.

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 [55] mostra que é sempre 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 uso do valor estático n permite obter um código mais eficiente. Um avaliador parcial para a linguagem C deveria produzir um programa residual como o exibido a seguir, ao especializar a função Power em relação a $n = 5$:

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

Avaliadores parciais já foram desenvolvidos com sucesso para linguagens de paradigma imperativo [2, 3, 66], funcional [54, 60, 11, 68] e lógico [46, 78, 69].

A avaliação parcial de programas é utilizada em diversas áreas da computação. Para que sua utilização valha a pena, deve ser levado em conta o custo de se produzir um programa especializado, em relação a um determinado parâmetro de entrada, é compensado pelo ganho de velocidade na execução. Se uma das entradas varia com freqüência bem menor que as outras, o custo de produção do programa especializado é diluído em um conjunto de execuções. Esse é o caso mais indicado para se utilizar a avaliação parcial. Entretanto,

muitas outras situações podem se valer das técnicas apresentadas neste capítulo. Veremos isso na Seção 3.2, que apresenta diversas aplicações de avaliação parcial.

A principal técnica utilizada para produzir programas especializados é designada *Especialização Polivariante*. O programa é visto como um grafo, onde os vértices são *pontos de programa*, conectados por *arestas de fluxo de controle*. Em linguagens de paradigma imperativo, os pontos de programa são geralmente blocos básicos e as arestas de fluxo de controle são os desvios de fluxo. Uma execução de um programa é vista como uma seqüência de estados, onde cada estado é definido por um ponto de programa e o valor das variáveis naquele ponto. O processo consiste em gerar todos os estados alcançáveis pelo programa, usando o valor da entrada conhecida (entrada estática). Em seguida, cada estado gerado dá origem a um bloco no programa residual, resultado da computação de toda informação estática do estado. A técnica é chamada de polivariante porque um mesmo bloco do programa original pode dar origem a diversos blocos no programa residual.

A especialização polivariante permite duas abordagens distintas: especialização *online* e especialização *offline*. Na abordagem *online*, a especialização é executada em um único passo. Os métodos *offline* dividem o processo em duas fases, onde a primeira é chamada *análise de tempo de definição* (BTA, do inglês *binding time analysis*), e a segunda é a especialização propriamente dita, seguindo as anotações produzidas pela BTA.

Na Seção 3.3, apresentamos a especialização polivariante e discutimos as diferenças e vantagens das abordagens *online* e *offline*. Uma linguagem simples de fluxograma, denominada FCL, é definida e utilizada nos exemplos ao longo de toda a seção.

A Seção 3.4 traz alguns exemplos de avaliação parcial de programas, usando as técnicas introduzidas na Seção 3.3. Todos os exemplos são escritos na linguagem FCL e a especialização é realizada usando a abordagem *offline*.

A compilação e geração de compiladores são algumas das aplicações mais importantes de avaliação parcial, assim esse assunto é discutido separadamente na Seção 3.5. Uma definição mais formal de avaliação parcial é exibida, servindo de base para a apresentação das *Três Projeções de Futamura*. Essas projeções são equações que mostram como, usando especialização de interpretadores, é possível realizar compilação dirigida por semântica. A auto-aplicação do avaliador parcial permite ainda geração de compiladores e geração de geradores de compiladores. Finalmente, é apresentada uma abordagem diferente para a avaliação parcial, denominada abordagem de *extensões de geração*. São apresentados argumentos que defendem a utilização dessa segunda abordagem quando se pretende realizar geração de compiladores para linguagens estaticamente tipadas.

A Seção 3.6 enumera algumas fontes de referência adicionais sobre avaliação parcial de programas. As conclusões deste capítulo são reunidas na Seção 3.7.

3.2 Aplicações de Avaliação Parcial

Esta seção apresenta aplicações da avaliação parcial de programas em diversas áreas.

3.2.1 Engenharia de Software

Dois objetivos importantes na produção de programas são geralmente conflitantes:

- construir programas genéricos e modulares, facilitando o reuso de código e manutenção do mesmo;
- construir programas os mais eficientes possíveis.

O preço de se utilizar módulos genéricos e fortemente independentes é geralmente a perda da eficiência. Muito tempo pode ser gasto em chamadas de funções, passagem de parâmetros, construção de estruturas complexas para satisfazer ao formato independente das interfaces, testes de valores que na realidade são constantes etc.

Embora se espere que um compilador eficiente execute a "propagação de constantes" em tempo de compilação, isso nem sempre é verdade quando se trata de propagação entre diferentes procedimentos. Além disso, a expansão de *loops* (*loop unrolling*) baseada em dados constantes é ainda mais rara.

Avaliação parcial pode ser utilizada para minimizar o impacto negativo da modularidade sobre a eficiência de programas. Mesmo não existindo dados estáticos para servir de base para a especialização, é possível identificar várias estruturas estáticas dentro de um programa, agrupar módulos dentro de um só, expandir chamadas de funções, expandir *loops* dependentes de constantes etc. O resultado é um programa inadequado à leitura ou modificação, mas muito mais eficiente.

3.2.2 Parâmetros com Freqüências de Variação Diferentes

Avaliação parcial também pode ser de grande valia em situações como a descrita a seguir:

- uma função $f(x, y)$ deve ser computada para diversos pares diferentes (x, y);
- o valor de x muda com menos freqüência que o de y; e
- uma parte significativa da computação de f depende apenas de x.

Um exemplo onde as condições acima são satisfeitas é o método usado em computação gráfica conhecido como *ray tracing*, para renderização de figuras. Esse método consiste em calcular o caminho de raios de luz entre vários pontos de uma cena que será exibida.

O algoritmo geral recebe duas entradas, uma cena e um raio de luz. Na exibição de uma figura, a cena, que é uma coleção de objetos de três dimensões, não é alterada enquanto a seqüência de raios é traçada. A avaliação parcial do algoritmo de *ray tracing* em relação a uma cena específica resulta em um procedimento eficiente para traçar raios para aquela cena [5]. O tempo gasto para se construir o algoritmo especializado é compensado pela eficiência obtida, uma vez que o mesmo será utilizado inúmeras vezes para diferentes raios. Experiências realizadas por Mogensen mostraram um *ray tracer* especializado que era de 8 a 12 vezes mais rápido que o original [67].

3.2.3 Problemas de Natureza Interpretativa

Problemas que tenham natureza interpretativa constituem outra classe que pode se beneficiar imensamente da avaliação parcial. Esses problemas envolvem geralmente o uso de uma linguagem que permite ao usuário especificar o formato da entrada e parâmetros especiais. Alguns exemplos são: simulação de circuitos, redes neuronais, casamento de padrões, problemas com entradas que são tabelas especificando transições.

Simuladores de circuitos recebem a descrição de um circuito elétrico como entrada, constroem equações diferenciais que descrevem seu comportamento e usam métodos numéricos para resolver essas equações. A especialização de um simulador geral em relação a um circuito específico gera um programa eficiente para esse circuito. Um ganho substancial em velocidade pode ser obtido [9].

O treinamento de redes neuronais geralmente é um processo muito caro, dispendendo um tempo longo de computação. Do ponto de vista da avaliação parcial, esse problema é semelhante ao anterior: um simulador geral pode ser especializado em relação a uma dada topologia de rede.

Um algoritmo de casamento de padrões em textos recebe como entradas uma seqüência de caracteres definindo um padrão e um texto onde esse padrão deverá ser pesquisado. A avaliação parcial do algoritmo em relação a um padrão específico resulta em um programa que implementa um autômato finito determinístico que executa as ações necessárias para a identificação desse padrão em qualquer texto. Uma experiência interessante de Consel e Danvy mostra a geração de um algoritmo equivalente ao método de Knuth, Morris e Pratt (KMP) a partir de algoritmo de casamento de padrões ingênuo (força bruta) [23].

Análise léxica e sintática podem ser implementadas usando-se algoritmos que seguem tabelas de transições. Essas tabelas associam, a um dado estado e um dado caractere de entrada, um novo estado e uma ação semântica correspondente. A especialização desses algoritmos em relação a tabelas específicas resulta na "compilação" da tabela diretamente para um código executável, muito mais rápido.

3.2.4 Compilação e Geração de Compiladores

Finalmente, um dos usos mais importantes de avaliação parcial é na compilação e geração de compiladores dirigida por semântica.

A semântica de uma linguagem de programação L pode ser dada por um interpretador para programas escritos em L. A avaliação parcial de um interpretador em relação a um programa específico resulta na compilação desse programa para a linguagem dos programas produzidos pelo avaliador parcial. A avaliação parcial do próprio avaliador em relação a um interpretador específico resulta em um compilador. A avaliação parcial do avaliador com respeito a si próprio resulta em um gerador de compiladores.

Os conceitos envolvidos neste exemplo são um pouco mais complexos que os dos exemplos anteriores, assim serão explicados em detalhe na Seção 3.5.

3.3 Técnicas de Avaliação Parcial

Para apresentar as técnicas empregadas na avaliação parcial de programas, vamos utilizar uma linguagem de fluxograma denominada FCL (*Flowchart Language*) [36]. Sendo bastante simples, essa linguagem é ideal para apresentarmos os conceitos básicos de avaliação parcial.

Vamos utilizar as seguintes notações:

$[y_o y_1 \ldots y_j]$ representa uma lista de tamanho $j+1$, com elementos y_o, y_1, $\ldots$ y_j.

$(x_o x_1 \ldots x_i)$ representa uma tupla de $i+1$ componentes; o k-ésimo componente, $0 \leq k \leq i$, é x_k.

3.3.1 Linguagem de Fluxograma FCL

A linguagem FCL é uma linguagem simples de fluxograma que contém apenas comandos de atribuição, desvio condicional e incondicional, e retorno de valores. O código é dividido em blocos básicos, identificados por rótulos.

Um programa FCL inicia-se com a definição dos parâmetros de entrada, seguida do rótulo do primeiro bloco a ser executado, seguido de uma série de blocos básicos, onde cada um possui uma única entrada e única saída. Os blocos são constituídos por uma seqüência de comandos de atribuição possivelmente vazia, seguida de exatamente um comando de desvio ou comando `return`. A cada bloco está associado um rótulo diferente.

Os comandos de atribuição têm o formato `v := exp`, onde `v` é uma variável e `exp` uma expressão contendo operações que envolvem variáveis e constantes. Um comando de desvio pode ser incondicional (`goto label`) ou condicional (`if boolexp goto label1 else label2`). O comando `return exp` retorna o valor da expressão calculada, terminando o programa. Todas as variáveis não pertencentes à entrada são inicializadas com zero. A Figura 3.2 mostra a codificação em FCL da função Power, exibida na Figura 3.1.

Observe que um comando de desvio é obrigatório ao final de todo bloco que não termine com o comando `return`, mesmo que o desvio seja para o bloco subseqüente no código. A execução do programa sempre é finalizada com um comando `return`.

A execução de um programa FCL pode ser representada por uma seqüência de estados. Cada estado pode ser representado por um par (l, σ), onde l é um ponto do programa indicado por um rótulo e σ é o valor das variáveis antes da execução do comando indicado por l. A execução da função Power da Figura 3.2, com $n = 2$ e $x = 5$, é apresentada na Figura 3.3. Um novo rótulo, não existente no programa, foi introduzido para representar o término da execução e retorno de um valor: $\langle halt, 25 \rangle$.

Na representação escolhida para execuções de FCL, o próximo comando a ser executado em um estado é sempre o primeiro comando de um bloco básico. Essa abordagem é adequada aos nossos propósitos de utilizar FCL para explicar o processo de avaliação parcial de programas, como veremos mais adiante. Outra alternativa seria adotar uma granularidade mais fina, com o estado representando cada comando do programa.

```
(n, x)
(b0)
b0:     p := 1
        goto b1
b1:     if n > 0 goto b2 else b5
b2:     if n % 2 = 0 goto b3 else b4
b3:     x := x * x
        n := n / 2
        goto b1
b4:     p := p * x
        n := n - 1
        goto b1
b5:     return p
```

Figura 3.2: Codificação em FCL da função Power.

	$(b0, [n \mapsto 2, x \mapsto 5, p \mapsto 0])$
passo 1:	$(b1, [n \mapsto 2, x \mapsto 5, p \mapsto 1])$
passo 2:	$(b2, [n \mapsto 2, x \mapsto 5, p \mapsto 1])$
passo 3:	$(b3, [n \mapsto 2, x \mapsto 5, p \mapsto 1])$
passo 4:	$(b1, [n \mapsto 1, x \mapsto 25, p \mapsto 1])$
passo 5:	$(b2, [n \mapsto 1, x \mapsto 25, p \mapsto 1])$
passo 6:	$(b4, [n \mapsto 1, x \mapsto 25, p \mapsto 1])$
passo 7:	$(b1, [n \mapsto 0, x \mapsto 25, p \mapsto 25])$
passo 8:	$(b5, [n \mapsto 0, x \mapsto 25, p \mapsto 25])$
passo 9:	$(\langle halt, 25\rangle, [n \mapsto 0, x \mapsto 25, p \mapsto 25])$

Figura 3.3: A execução da função Power com $n = 2$ e $x = 5$.

3.3.2 Especialização Polivariante

A técnica de *especialização polivariante* para avaliação parcial de programas é utilizada em linguagens de diversos paradigmas diferentes. O programa é visto como um grafo, onde os vértices são *pontos de programa*, conectados por *arestas de fluxo de controle*. Na linguagem FCL, os pontos de programa e as arestas de fluxo de controle são, respectivamente, os *blocos básicos rotulados* e os *desvios*; em uma linguagem funcional, eles seriam as *funções definidas* e as *chamadas de funções*.

Essa técnica é chamada de especialização polivariante porque um único ponto de programa do programa original pode dar origem a vários pontos de programa no programa especializado. Para entender o processo, considere como a execução exibida na Figura 3.3 teria que ser alterada se apenas parte dos dados de entrada fosse fornecida. Intuitivamente, podemos prever que alguns dos valores das variáveis representadas em cada estado seriam desconhecidos. Algumas das expressões, atribuições, e desvios poderiam ser computados, se dependessem apenas dos dados conhecidos, chamados de *entrada estática*. Outras estruturas do programa não poderiam ser computadas, se dependessem dos valores não conhecidos, chamados de *entrada dinâmica*. O processo procura estabelecer todos os estados alcançáveis, utilizando apenas os dados estáticos. Em seguida, constrói os blocos básicos do programa especializado. Cada bloco é derivado de um estado alcançado: o bloco construído a partir de (l, σ) é o resultado da computação de toda informação estática do bloco l, utilizando os valores conhecidos que aparecem em σ.

Podemos ver agora que o formato escolhido para a representação da execução de um programa FCL é adequada ao processo descrito acima. Cada estado da execução está associado ao início de um bloco básico, coincidindo com os pontos de programa utilizados no método de especialização polivariante.

Resumindo, o processo é composto de três passos:

1. Determinação de todos os estados alcançáveis (*reachable states*): iniciando no estado inicial (l_0, σ_0), onde σ_0 contém apenas os dados conhecidos, reunir todos os estados alcançáveis no conjunto S.

2. Especialização dos pontos de programa (*program point specialization*): para cada $(l_i, \sigma_i) \in S$, inserir no programa residual um novo bloco, resultado da especialização de l_i em relação aos valores de σ_i.

3. Compressão das transições (*transition compression*): otimização do programa residual por meio da fusão de alguns blocos; blocos compostos por apenas um desvio incondicional estão entre os candidatos a serem eliminados.

A maioria dos avaliadores parciais executa esses três passos em paralelo.

Como visto, o processo inicia com a divisão dos dados de entrada em conhecidos (estáticos) e não conhecidos (dinâmicos). Para determinar se os objetos restantes do programa são estáticos ou dinâmicos, existem ainda duas abordagens diferentes: métodos *online* e *offline*. Nos métodos *online*, os valores são classificados como estáticos ou dinâmicos durante a especialização, enquanto que, nos métodos *offline*, uma análise do programa é feita antes de iniciar o processo de especialização, determinando a divisão. A seguir,

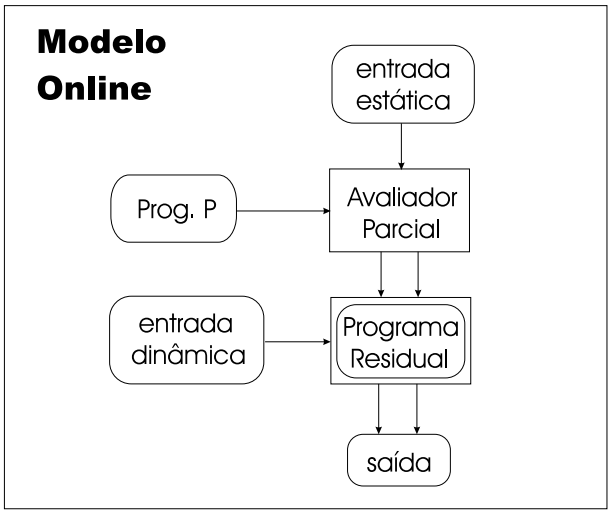


Figura 3.4: Esquema de uso das técnicas *online*.

discutimos essas duas abordagens e fornecemos exemplos da aplicação da especialização polivariante usando cada um deles.

3.3.3 Métodos *Online*

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. A Figura 3.4 exibe um esquema do uso dessas técnicas. Programas são representados dentro de retângulos, e dados são representados dentro de retângulos com bordas arredondadas. Setas simples indicam entrada de um programa e setas duplas indicam sua saída.

Nesta seção, vamos mostrar a abordagem *online* através de um exemplo, seguindo os três passos da especialização polivariante. Como dissemos antes, esses três passos são comumentes executados em paralelo. Optamos por mostrar a execução de cada passo separadamente nos primeiros exemplos, por questões didáticas.

Determinação dos Estados Alcançáveis

Para demonstrar o processo, vamos voltar à execução do programa Power, exibida na Figura 3.3. Desta feita, vamos considerar apenas que o valor de n é 2 (entrada estática), mas o valor de x não é conhecido. O símbolo D vai ser utilizado para representar o valor de x (entrada dinâmica). Sendo assim, o estado inicial terá $b0$ como ponto de programa e $[n \mapsto 2, x \mapsto D, p \mapsto 0]$ como valor das variáveis. O valor inicial de p, embora não seja uma entrada conhecida, será 0, que é o valor *default* das variáveis. Apenas as entradas dinâmicas são inicializadas com D.

Mais tarde, quando apresentarmos um algoritmo para especialização *online*, veremos que é necessário elaborar um pouco mais essa representação: o valor de uma variável será denotado por um par $(type, val)$, onde $type$ será D ou S (dinâmico ou estático) e val será o valor real da variável, se for estática.

A execução da função Power, com valor n igual a 2, terá o seguinte formato:

$(b0, [n \mapsto 2, x \mapsto D, p \mapsto 0])$
passo 1: $(b1, [n \mapsto 2, x \mapsto D, p \mapsto 1])$
passo 2: $(b2, [n \mapsto 2, x \mapsto D, p \mapsto 1])$
passo 3: $(b3, [n \mapsto 2, x \mapsto D, p \mapsto 1])$
passo 4: $(b1, [n \mapsto 1, x \mapsto D, p \mapsto 1])$
passo 5: $(b2, [n \mapsto 1, x \mapsto D, p \mapsto 1])$
passo 6: $(b4, [n \mapsto 1, x \mapsto D, p \mapsto 1])$
passo 7: $(b1, [n \mapsto 0, x \mapsto D, p \mapsto D])$
passo 8: $(b5, [n \mapsto 0, x \mapsto D, p \mapsto D])$
passo 9: $(halt, [n \mapsto 0, x \mapsto D, p \mapsto D])$

O que acontece a cada passo:

Passo 1: No bloco $b0$, o valor de p foi aterado para 1, e o próximo ponto de programa calculado é $b1$.

Passo 2: Como n é conhecido, a condição do desvio pôde ser computada, definindo que o próximo ponto de programa é $b2$.

Passo 3: De maneira similar ao passo anterior, o próximo ponto de programa calculado é $b3$.

Passo 4: O valor de n é alterado para 1, mas como x não é conhecido, $x * x$ não pôde ser calculado.

Passo 5: De maneira equivalente ao passo 2, o próximo ponto de programa é $b2$.

Passo 6: Como o valor de n é ímpar, o próximo ponto de programa é $b4$.

Passo 7: A expressão $p * x$ não pôde ser calculada, pois x não é conhecido, assim o valor de p é alterado para D. O valor de n é alterado para 0.

Passo 8: A condição do desvio falhou desta vez, assim o próximo ponto de programa é $b5$.

Passo 9: Fim do programa. O valor de retorno não pôde ser calculado.

A execução do programa baseada nos valores conhecidos determina todos os estados alcançáveis a partir do estado inicial. Dois aspectos que valem a pena ser mencionados não ocorrem no exemplo mostrado nesta seção:

- Se for gerado um estado que já foi produzido anteriormente na execução, o algoritmo de determinação dos estados alcançáveis não precisa repetir todo o processo. Esse caso irá corresponder a um *loop* no programa residual.
- No exemplo exibido, todos os testes dos desvios condicionais dependiam apenas dos valores estáticos, assim foi sempre possível identificar qual seria o próximo bloco a ser executado. Se a condição depender de valores dinâmicos, não teremos como computá-la. Nesse caso, o algoritmo deve gerar os estados alcançáveis levando em conta os dois blocos referenciados no desvio condicional.

Especialização dos Pontos de Programa

Na execução exibida na seção anterior, pode-se ver que um mesmo ponto de programa aparece mais de uma vez, com diferentes valores para as variáveis. Por exemplo, $b1$ aparece com os seguintes valores para as variáveis:
$[n \mapsto 2, x \mapsto D, p \mapsto 1]$, $[n \mapsto 1, x \mapsto D, p \mapsto 1]$, $[n \mapsto 0, x \mapsto D, p \mapsto D]$.

Cada instância irá gerar um bloco básico especializado diferente, no programa residual. Blocos diferentes recebem rótulos diferentes, assim os rótulos serão também especializados. Por motivo de simplicidade, vamos utilizar o par (l_i, σ_i) para denotar cada rótulo diferente. Por exemplo, a primeira instância do bloco $b1$ terá um rótulo como o seguinte: $(b1, [2, D, 1])$. Os rótulos referenciados nos comandos de desvio condicional e incondicional deverão também ser alterados.

O programa residual terá o seguinte formato, antes das otimizações serem conduzidas:

```
(x)
((b0, [2, D, 0]))
(b0, [2, D, 0]):   goto (b1, [2, D, 1])
(b1, [2, D, 1]):   goto (b2, [2, D, 1])
(b2, [2, D, 1]):   goto (b3, [2, D, 1])
(b3, [2, D, 1]):   x := x * x
                   goto (b1, [1, D, 1])
(b1, [1, D, 1]):  goto (b2, [1, D, 1])
(b2, [1, D, 1]):  goto (b4, [1, D, 1])
(b4, [1, D, 1]):  p := 1 * x
                  goto (b1, [0, D, D])
(b1, [0, D, D]):   goto (b5, [0, D, D])
(b5, [0, D, D]):   return p
```

A seguir, exibimos uma explicação de como o código de cada um dos blocos especializados foi obtido:

$(b0, [2, D, 0])$**:** A atribuição `p := 1;` não depende de nenhuma informação dinâmica, assim pode ser completamente computada. A instrução `goto b1` vai aparecer no código residual, especializada em relação ao novo valor das variáveis, ou seja, $[2, D, 1]$.

$(b1, [2, D, 1])$**:** Sempre que a condição de um desvio puder ser totalmente computada, ela não precisa aparecer no código residual. Além disso, a informação pode ser usada para decidir qual dos dois blocos especializar. No caso deste bloco, a condição é verdadeira e um desvio especializado para o bloco $b2$ é inserido no código residual.

$(b2, [2, D, 1])$**:** De forma semelhante ao caso anterior, um desvio especializado para o bloco $b3$ é inserido no código residual.

$(b3, [2, D, 1])$**:** Sempre que um dos valores de uma operação não for conhecido, a operação é classificada como D, ou seja, dinâmica. A variável x tem valor D, assim a expressão $x * x$ é classificada também como D. Portanto, `x := x*x;` deve aparecer no código residual. Por outro lado, na operação $n/2$, os valores envolvidos são uma

variável estática e uma constante, assim a expressão pode ser calculada. E como n é estática, a atribuição `n := n/2;` vai ser computada, não aparecendo no código residual. Finalmente, um desvio especializado é produzido.

$(b1, [1, D, 1])$**:** Similar a $(b1, [2, D, 1])$.

$(b2, [1, D, 1])$**:** Similar a $(b2, [2, D, 1])$.

$(b4, [1, D, 1])$**:** A operação $p * x$ é classificada como D, pois x é D. Entretanto, a residualização envolve o cálculo dos componentes estáticos da operação, nesse caso, a variável p. O resultado é a expressão $1 * x$. Sempre que um valor estático aparece em um contexto dinâmico, aplicamos uma operação conhecida como *lift*. Dizemos que o valor estático 1, calculado a partir de p, foi *lifted* para aparecer no código residual. Na atribuição `p := p * x`, a expressão do lado direito foi classificada como D, assim p passa também a ser D e a atribuição é residualizada. O resto do bloco é simples, consistindo do cálculo do novo valor de n e a residualização do desvio. O valor das variáveis agora é $[0, D, D]$.

$(b1, [0, D, D])$**:** Similar a $(b1, [2, D, 1])$ e $(b1, [1, D, 1])$.

$(b5, [0, D, D])$**:** A expressão p é classificada como D. Um comando `return` deve sempre ser residualizado. Se a expressão a ser retornada for estática, deve sofrer *lift* para aparecer no código residual.

Compressão das Transições

O programa produzido na seção anterior contém vários blocos terminados por um desvio incondicional, alguns deles formados apenas pelo desvio. Esse fato ocorre com muita freqüência quando se utiliza o método de especialização polivariante, pois muitos comandos de blocos básicos do programa original não aparecem no programa residualizado.

A compresssão de transições consiste em substituir um desvio para um rótulo *pp* pelo código do bloco indicado por *pp*. Os benefícios são evidentes: o programa pode ficar bem mais eficiente, com a eliminação de muitos desvios no código. Se aplicarmos a compressão de transições no programa residual gerado na seção anterior, obteremos o resultado a seguir, onde os rótulos foram renomeados:

```
(x)
(b0)
b0:   x := x * x
      p := 1 * x
      return p
```

Se aplicarmos a compressão de transições indiscriminadamente, podemos ter dois tipos de problemas: duplicação de código e compressão infinita. A duplicação de código ocorre quando a compressão é aplicada a duas transições distintas para um mesmo ponto de programa. Quando o programa residual contém um *loop*, a compressão pode continuar indefinidamente.

Se a compressão de transições é executada como uma fase separada, após a geração do programa residual, a duplicação de código e compressão infinita podem ser mais facilmente evitadas. Um grafo de fluxo de controle do programa residual pode ser construído, e uma análise pode ser conduzida de forma a identificar as compressões seguras. Entretanto, a especialização polivariante gera com freqüência inúmeros blocos contendo desvios supérfluos, assim seria mais eficiente conduzir a compressão de transições durante o processo de especialização (*transition compression on the fly*).

Executar a compressão de transições durante a especialização pode tornar bem mais difícil a tarefa de identificar as compressões seguras. Uma estratégia simples é usar a compressão em todas as transições que não façam parte de um desvio condicional residual. Na realidade, isso seria o máximo que conseguiríamos com a linguagem FCL, uma vez que um desvio condicional não pode conter comandos a serem executados, diferente da estrutura de blocos usada pela maioria das linguagens imperativas.

A estratégia descrita ainda pode gerar duplicação de código, mas experiências relatadas indicam que é um problema mínimo [55]. O mais importante é que essa estratégia não produz uma compressão infinita, a não ser que o programa residual contenha um *loop* infinito que dependa apenas da entrada estática. Nesse caso, o programa original também entraria em *loop* infinito com os mesmos valores estáticos, independentemente dos valores dinâmicos utilizados.

Outras Otimizações

O programa residual gerado pela especilização polivariante pode sofrer ainda outras otimizações.

Usando Propriedades Algébricas A operação de multiplicação

```
p := 1 * x;
```

poderia ser otimizada para

```
p := x;
```

Entretanto, poucos avaliadores parciais implementam otimizações como essa. O esforço adicional necessário geralmente não é compensado pelos resultados alcançados.

Expandindo Expressões No código a seguir:

```
x := x * x;
p := 1 * x;
```

o primeiro comando de atribuição poderia ser expandido dentro do segundo:

```
p := 1 * x * x;
```

Dessa forMa, pode-se diminuir o número de comandos de atribuição. Entretanto, a utilização indiscriminada dessa técnica pode levar à duplicação indesejável de código. Por

```
pending := {(pp0,vs0)};
marked := {};
while (pending ≠ {}) do begin
    retire um elemento (pp,vs) de pending;
    marked := marked ∪ {(pp,vs)};
    bb := lookup (pp, program);
    (* bb é o bloco rotulado por pp no programa original *)
    code := newblock (pp,vs);
    (* cria novo bloco rotulado por (pp,vs) *)
    while (bb não estiver vazio) do begin
        command := first_command (bb);
        bb := rest (bb);
        case command of
            ...
    end;
    residual := extend (residual, code);
    (* adiciona bloco ao programa residual *)
end
```

Figura 3.5: MIX - Algoritmo para Especialização de Programas.

exemplo:

```
x := (a + b) * (c + d);
y := x + x;
```

poderia ser expandido para

```
y := ((a + b) * (c + d)) + ((a + b) * (c + d));
```

Uma análise que armazene o número de referências às variáveis é necessária para determinar se o código não irá ser duplicado, como no exemplo acima.

Especialização Online: Três Passos em Paralelo

Um algoritmo para realizar especialização de programas FCL (que na Seção 3.1 chamamos de `mix`), executando os três passos em paralelo, é exibido na Figura 3.5. O algoritmo tem como entradas o programa original (`program`), o rótulo do bloco inicial do programa ($\texttt{pp}_0$) e o valor inicial das variáveis ($\texttt{vs}_0$).

O valor de cada variável é representado por um par $(type, val)$, onde $type$ é S (estático) ou D (dinâmico). Se a variável for dinâmica, val contém uma expressão representando a própria variável; se for estática, val contém a constante associada. No exemplo desenvolvido nesta seção, a lista de valores iniciais $\texttt{vs}_0$ seria: $[n \mapsto (S, 2)\,,\ x \mapsto (D, x)\,,\ p \mapsto (S, 0)]$. Em métodos *online*, a utilização dos "*tags*" S e D para diferenciar valores estáticos e dinâmicos é essencial.

O conjunto `pending` contém os blocos do programa residual que ainda não foram gerados. Esses blocos são identificados por um par (pp, vs), onde pp é um rótulo do programa

original e *vs* é uma lista de valores para as variáveis. O conjunto `marked` é utilizado para armazenar os rótulos dos blocos gerados no programa residual, para evitar o processamento de um bloco que já foi gerado.

O *loop* mais interno processa cada comando FCL do bloco corrente. Esses comandos podem alterar o estado *vs* das variáveis e também gerar novos blocos a serem inseridos em `pending`. O código para processar cada comando é exibido na Figura 3.6.

Primeiro discutiremos o processamento dos comandos de desvio. A definição de FCL impõe que qualquer comando de desvio sempre esteja posicionado no final de um bloco. Se o comando é um desvio incondicional `goto pp'`, a compressão da transição será realizada. No algoritmo, isso é feito pelo comando `bb := lookup (pp', program)`, lembrando que a variável `bb` contém o restante do bloco que está sendo processado em um dado momento. Nesse caso, os resultados da especialização do bloco corrente e de `pp'` aparecerão no mesmo bloco do programa residual.

Para processar um desvio condicional `if exp goto pp' else pp''`, o primeiro passo é a avaliação da condição `exp,` executada por `eval(exp,vs)`. Essa função avalia uma expressão do programa original, usando uma lista de valores para as variáveis. O valor retornado é um par `(type,val)`, onde `type` pode ser S (estático) ou D (dinâmico). Se `type` é S, a expressão não depende de valores dinâmicos e `val` contém o valor constante resultado de sua avaliação. Se `type` é D, a expressão depende de valores dinâmicos e deve ser residualizada. Nesse caso, `val` contém o código da expressão especializado em relação às variáveis de `vs`.

Se a avaliação de `exp` no desvio condicional `if exp goto pp' else pp''` resultar em um valor estático, este será a constante TRUE ou FALSE. Se for TRUE, é executada uma compressão sobre a transição `pp'`, caso contrário, sobre `pp''`. Esse é um processo análogo ao que é conduzido quando o desvio é incondicional. Se o resultado da avaliação da condição for D, significa que ela depende de valores dinâmicos. Nesse caso, a compressão de transição não é realizada. Um comando de desvio condicional é gerado no final do bloco corrente. Esse comando é construído utilizando a condição especializada armazenada em `val`. Além disso, dois novos blocos podem ser inseridos em `pending`, se ainda não foi gerado código para eles. O algoritmo usa `marked` para determinar se código já foi gerado para os blocos em questão.

Na parte inicial do código exibido na Figura 3.6, pode-se ver o processamento dos comandos de atribuição e retorno de valor. Em um comando `return`, primeiro a expressão é processada. Se resultar em um valor estático, ela deve sofrer *lift* para aparecer no código residual, uma vez que todo comando `return` é residualizado. Em um comando de atribuição, primeiro o lado direito é processado. Se o resultado for um valor estático, o valor da variável é modificado e nenhum código é gerado. Se for dinâmico, a expressão especializada é utilizada na construção do comando de atribuição que irá aparecer no programa residual; além disso, a variável passa a ser classificada como dinâmica. A notação utilizada para modificação do valor de uma variável é `vs[`$X \mapsto$ `(type,val)]`, que significa: retorne uma nova lista `vs'` idêntica a `vs`, com exceção do valor da variável X, que é alterado para `(type,val)`.

```
case command of

(* se for comando return *)
return exp :
begin
    (type,val) := eval (exp, vs);
    if (type = S) then val = lift (val);
    code := extend (code, return val);
end

(* se for comando de atribuição *)
X := exp :
begin
    (type,val) := eval (exp, vs);
    vs := vs[X ↦ (type,val)];
    if (type = D) then
        code := extend (code, X := val);
end;

(* se for desvio incondicional *)
goto pp' :
    (* compressão da transição *)
    b := lookup (pp', program);

(* se for desvio condicional *)
if exp goto pp' else pp'' :
begin
    (type,val) := eval (exp, vs);
    if (type = S) then (* compressão da transição *)
        if (val = TRUE) then
            bb := lookup (pp', program);
        else (* val = FALSE *)
            bb := lookup (pp'', program);
    else begin (* type = D *)
        pending := pending ∪ ({(pp',vs)}\ marked);
        pending := pending ∪ ({(pp'',vs)}\ marked);
        code := extend (code,
            if val goto (pp',vs) else (pp'',vs) );
    end
end;
```

Figura 3.6: Método *Online* para Especialização de Comandos FCL.

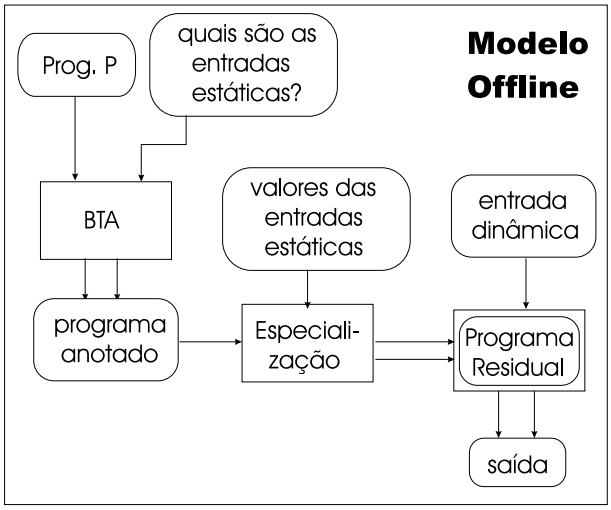


Figura 3.7: Esquema de uso das técnicas *offline*.

3.3.4 Métodos *Offline*

Vimos que, nos métodos *online*, a determinação dos valores estáticos e dinâmicos é realizada junto com a especialização. Na abordagem *offline*, por outro lado, o processo de especialização é geralmente dividido em duas fases.

A primeira fase de um método *offline* é denominada BTA (*binding time analysis* ou *análise de tempo de definição*). Como nos métodos *online*, o processo começa pela determinação de que partes da entrada do programa são conhecidas (estáticas) e que partes não são conhecidas (dinâmicas). Entretanto, o algoritmo de BTA não utiliza os valores especificados para as entradas estáticas. É feita uma análise sobre o programa, determinando quais variáveis dependem direta ou indiretamente das entradas estáticas e dinâmicas. Com isso, é produzida uma *divisão* de todas as variáveis nessas duas classes. Na maioria das vezes, essa informação é utilizada também para classificar cada estrutura do programa (comandos, operações) como estática ou dinâmica. Assim, BTA geralmente produz um *programa anotado* que identifica exatamente quais estruturas vão ser computadas e quais vão ser residualizadas.

A segunda fase de um método *offline* é a especialização propriamente dita. Como nos métodos *online*, a especialização gera os estados alcançáveis, especializa os pontos de programa e realiza a compressão das transições. Dados os valores das variáveis estáticas de entrada, o algoritmo segue estritamente as anotações geradas pela BTA para produzir o programa residual. Ao contrário dos métodos *online*, não é necessário determinar se uma operação é estática ou dinâmica, cada vez que ela for analisada.

A Figura 3.7 exibe um esquema do funcionamento dos métodos offline para avaliação parcial. Nas seções seguintes, vamos mostrar a execução dos processos de geração de estados alcançáveis, especialização de pontos de programa e compressão de transições usando novamente a função Power da Figura 3.2.

Análise de Tempo de Definição

Suponha que a função Power deva ser especializada em relação a n, o primeiro parâmetro. Essa é a única informação necessária para executar a BTA. De modo diverso dos métodos *online*, não será utilizado, por enquanto o valor fornecido para a entrada estática n.

O objetivo inicial é descobrir quais variáveis utilizadas no programa são estáticas e dinâmicas, isto é, computar uma *divisão* das variáveis. Em seguida, um programa anotado é gerado.

A computação da divisão das variáveis pode seguir dois métodos:

1. iteração até atingir um ponto fixo;
2. resolução de restrições (*constraint solving*).

O primeiro método é o mais simples e utilizado pela maioria dos avaliadores parciais mais antigos. Nesta seção utilizaremos apenas esse método, que executa diversos passos sobre o programa, até atingir um ponto fixo na divisão das variáveis. Referências sobre análise de tempo de definição usando resolução de restrições podem ser encontradas em [11, 12, 14].

Se a função Power, do Figura 3.2, vai ser especializada em relação a n, o algoritmo de iteração até atingir um ponto fixo é disparado com a especificação $[n \mapsto S, x \mapsto D]$, indicando que n é estático (S) e x é dinâmico (D). A seguinte *divisão inicial* é computada:

$$\Delta = [n \mapsto S, x \mapsto D, p \mapsto S]\,,$$

que casa com a especificação de entrada e determina que as demais variáveis são estáticas. Podemos assumir que as variáveis não pertencentes à entrada são inicialmente estáticas, pois todas as variáveis são inicializadas com 0 em FCL.

O algoritmo executa iterações seqüenciais sobre o programa, modificando a divisão, até que mais nenhuma modificação seja processada. O princípio utilizado para as modificações é a *congruência*: qualquer variável que dependa de uma variável dinâmica deve também ser classificada como dinâmica. No caso de FCL, a dependência é dada pelos comandos de atribuição: `V := exp` indica que `V` depende de quaisquer variáveis que apareçam em `exp`.

Assim, uma alteração na divisão só ocorre se uma variável que antes era classificada com estática passar a ser dinâmica. O número de variáveis é finito, logo garante-se que o processo sempre termina. Essa estratégia é conservadora, pois uma variável é considerada dinâmica se o for em qualquer ponto do programa. É conhecida por *divisão uniforme*, pois vale para o programa inteiro. Mais tarde veremos como se pode definir uma divisão diferente para cada ponto do programa (*pointwise division*) e várias divisões para um mesmo ponto (*polyvariant division*).

Voltando ao exemplo, a primeira iteração do algoritmo determina que a variável p deve ser dinâmica, uma vez que depende de x no comando `p := p * x`. Na segunda iteração, nenhuma mudança é realizada, assim o algoritmo termina com a seguinte divisão:

$$\Delta' = [n \mapsto S, x \mapsto D, p \mapsto D]\,.$$

```
(n, x)
(b0)
b0:     p := 1
        goto b1
b1:     if n > 0 goto b2 else b5
b2:     if n % 2 = 0 goto b3 else b4
b3:     x := x * x
        n := n / 2
        goto b1
b4:     p := p * x
        n := n - 1
        goto b1
b5:     return p
```

Figura 3.8: Programa Anotado.

O próximo passo é gerar um programa anotado. As estruturas do programa que dependem das variáveis dinâmicas são marcadas como *residualizávies*. Se um dos operandos de uma operação é dinâmico, a operação é classificada como dinâmica. Um comando de atribuição é dinâmico se a variável do lado esquerdo o for.

Na Figura 3.8, é exibido o código anotado da função Power, a partir da divisão computada acima.. A codificação é feita usando-se uma linguagem de dois níveis [74, 73]. As estruturas dinâmicas aparecem sublinhadas.

As variáveis não precisam ser anotadas porque sua classificação pode ser obtida diretamente da divisão computada. Todos os comandos de atribuição cujo lado esquerdo é p ou x são anotados como residualizáveis, pois essas variáveis são dinâmicas. No primeiro bloco, a constante 1 é anotada, indicando que irá aparecer no código residual. Como visto na Seção 3.3.3, sempre que um valor estático aparece em um contexto dinâmico, ele sofre um *lift*. Assim, a anotação da constante 1 é equivalente a escrever `lift(1)`, mas neste caso a operação de *lift* está *compilada* no código anotado. Por enquanto, vamos deixar a compressão de transições de lado, assim todos os `goto` são anotados.

Geração dos Estados Alcançáveis

Esse é o primeiro passo da especialização, seguindo a geração do programa anotado produzido na fase de BTA. Lembramos novamente que os três passos da especialização são geralmente conduzidos paralelamente, mas neste exemplo vamos executá-los em seqüência.

A propriedade de congruência da divisão das variáveis garante que nenhuma variável estática depende de uma dinâmica. Desse modo, a representação dos estados pode ser mais simples que a empregada nos métodos *online*. Cada estado será representado apenas pelos valores das variáveis estáticas.

No nosso exemplo, o estado inicial será $(b0, [n \mapsto 2])$. A seqüência de estados gerada é exibida a seguir:

$(b0, [n \mapsto 2])$
$\rightarrow \quad (b1, [n \mapsto 2])$
$\rightarrow \quad (b2, [n \mapsto 2])$
$\rightarrow \quad (b3, [n \mapsto 2])$
$\rightarrow \quad (b1, [n \mapsto 1])$
$\rightarrow \quad (b2, [n \mapsto 1])$
$\rightarrow \quad (b4, [n \mapsto 1])$
$\rightarrow \quad (b1, [n \mapsto 0])$
$\rightarrow \quad (b5, [n \mapsto 0])$
$\rightarrow \quad (halt, [n \mapsto 0])$

Para gerar a seqüência de estados, basta seguir as anotações. As construções sublinhadas são ignoradas, uma vez que não contribuem para a determinação dos valores estáticos.

Especialização dos Pontos de Programa

Como nos métodos *online*, para cada estado (l_i, σ_i), uma versão especializada do bloco l_i é criada, usando os valores das variáveis estáticas de σ_i. A diferença é que não é necessário verificar se cada operação é S ou D, uma vez que isso já foi estabelecido pelas anotações.

Por exemplo, no bloco

```
b3:  x := x * x
     n := n / 2
     goto b1
```

todos os componentes do primeiro comando de atribuição estão sublinhados, assim são copiados diretamente para o código residual. O segundo comando, ao contrário, gera uma atualização do valor de n. Finalmente, o comando de desvio é copiado para o código residual. O programa residual é mostrado abaixo:

```
(x)
((b0, [2]))
(b0, [2]):   P := 1
             goto (b1, [2])
(b1, [2]):   goto (b2, [2])
(b2, [2]):   goto (b3, [2])
(b3, [2]):   x := x * x
             goto (b1, [1])
(b1, [1]):   goto (b2, [1])
(b2, [1]):   goto (b4, [1])
(b4, [1]):   p := p * x
             goto (b1, [0])
(b1, [0]):   goto (b5, [0])
(b5, [0]):   return p
```

Compressão de Transições

A compressão de transições é feita da mesma maneira que nos métodos *online*. Assim, o programa residual terá o seguinte formato, após as otimizações dos desvios:

```
(x)
((b0))
(b0):  P := 1
       x := x * x
       p := p * x
       return p
```

Especialização Offline: Três Passos em Paralelo

Um algoritmo para avaliação parcial de programas usando um método *offline* tem o mesmo formato geral proposto para os métodos *online*, exibido na Figura 3.5. A diferença reside em como os comandos são tratados, refletindo a opção entre:

- calcular o caráter estático ou dinâmico de cada componente de modo *online*, isto é, durante a execução da especialização; ou
- produzir um programa anotado com informações de análise de tempo de definição e depois simplesmente seguir essas anotações, na especialização.

No método *online*, o algoritmo da Figura 3.5 tinha como entradas o programa original (`program`), o rótulo do bloco inicial do programa ($\texttt{pp}_0$) e o valor inicial das variáveis ($\texttt{vs}_0$). O método *offline* precisa dessas mesmas entradas, e de uma quarta, que é a divisão das variáveis computada pela BTA. Além disso, `program` é o programa anotado produzido pela BTA, e não mais o programa original.

O exemplo estudado nesta seção, que foi a especialização *offline* da função Power, nos mostrou que é necessário representar apenas o valor das variáveis estáticas em cada estado. Desse modo, elimina-se a necessidade de utilizar *tags* para identificar se um valor de uma variável é estático ou dinâmico, durante a especialização. O estado inicial $\texttt{vs}_0$ das variáveis, no caso da função Power especializada em relação a n, será $[n \mapsto 2]$.

Como fizemos na Seção 3.3.3, o código *offline* para processar cada comando FCL (`return`, atribuição, desvio condicional e incondicional) é apresentado separadamente do *loop* principal do algoritmo de especialização. Esse código pode ser observado na Figura 3.9.

As funções utilizadas na Figura 3.9 para avaliar expressões são `eval` e `reduce`. A função `eval` só é aplicada a expressões classificadas pela BTA como estáticas, retornando uma constante resultante da avaliação dessas expressões. A função `reduce`, por outro lado, só é aplicada a expressões classificadas como dinâmicas. O resultado da aplicação de `reduce` a uma expressão é uma nova expressão, onde todas as operações estáticas foram computadas. Isso configura uma política completamente diferente da adotada na abordagem *online*, onde a avaliação de uma expressão poderia resultar em um valor estático ou dinâmico, necessitando da propagação de valores com *tags*.

```
case command of

(* se for comando return *)
return exp :
    code := extend (code, return reduce(exp,vs) );

(* se for comando de atribuição *)
 X := exp :
    if (X foi classicada como estática) then
        vs := vs[X ↦ eval(exp,vs)];
    else
        code := extend (code, X := reduce(exp,vs) );

(* se for desvio incondicional *)
goto pp' :
    (* compressão da transição *)
    b := lookup (pp', program);

(* se for desvio condicional *)
if exp goto pp' else pp'' :
    if (exp foi classificada como estática) then
        if (eval(exp,vs) = TRUE) then
            (* compressão da transição *)
            bb := lookup (pp', program);
        else
            (* compressão da transição *)
            bb := lookup (pp'', program);
    else begin
        (* desvio condicional dinâmico *)
        pending := pending ∪ ({(pp',vs)}\ marked);
        pending := pending ∪ ({(pp'',vs)}\ marked);
        code := extend (code,
            if reduce(exp,vs) goto (pp',vs) else (pp'',vs) );
    end
```

Figura 3.9: Método *Offline* para Especialização de Comandos FCL.

3.3.5 Comparação Entre Métodos *Online* e *Offline*

Para tecer comparações entre as duas abordagens propostas para avaliação parcial, vamos observar os programas residuais produzidos para a função Power, nas Seções 3.3.3 e 3.3.4.

Ambas as abordagens levaram à produção de nove estados diferentes, mas o método *online* possibilitou uma maior exploração do valor estático da variável p. Recordando o raciocínio desenvolvido, o comando de atribuição

```
p := 1;
```

foi residualizado pelo método *offline*, mas foi computado pelo método *online* e não apareceu no programa residual. Isso aconteceu porque a variável p foi identificada pela BTA como dependendo de valores dinâmicos em algum ponto do programa, assim o método *offline* classificou toda operação envolvendo p como dinâmica no programa anotado. O comando de atribuição

```
p := p * x;
```

também foi completamente residualizado pelo método *offline*, enquanto a especialização *online* pôde inferir o valor constante 1 a partir de p.

Os resultados alcançados, embora simples, mostram que os métodos *offline* são muitas vezes mais conservadores do que a abordagem *online*. Uma análise mais cuidadosa revela, entretanto, que muitas vantagens são conseguidas quando se utiliza a abordagem *offline*.

O que deve ser comparado quando se escolhe uma das abordagens é:

- ter que lidar com a propagação de *tags* que identificam se um valor é estático ou dinâmico, durante o processo de especialização; ou
- ter que produzir um programa anotado e realizar a especialização em duas fases.

Uma analogia com as políticas de verificação de tipos de linguagens de programação pode dar uma idéia clara das diferenças. Avaliação parcial *online* é análoga à verificação de tipos em tempo de execução, onde os valores posuem *tags* associados indicando o tipo. Avaliação parcial *offline*, por outro lado, é análoga à verificação de tipos estática.

Da mesma maneira que verificação de tipos dinâmica, especialização *online* tem a vantagem de ser mais flexível e menos conservadora. Pode se basear nos valores reais das variáveis para tomar decisões sobre o caráter estático ou dinâmico de uma construção do programa analisado. Na abordagem offline, a fase de BTA não tem acesso aos valores das variáveis, tendo que decidir a classificação baseada apenas nos tags S e D.

Da mesma maneira que a verificação de tipos estática, avaliação parcial *offline* tem a vantagem de ser mais eficiente e permite uma verificação mais fácil de propriedades do programa residual, antes que ele seja gerado. Uma vez que a divisão das variáveis é executada uma única vez, elimina-se o esforço adicional de se lidar com *tags* durante a especialização.

Os primeiros avaliadores parciais usavam sempre métodos *online*. As técnicas *offline* receberam um grande impulso quando se buscou a compilação e geração de compiladores usando avaliação parcial. Essas tarefas envolvem a auto-aplicação do avaliador parcial,

processo que introduz muitos problemas para a abordagem *online*, mas que puderam ser resolvidos de maneira satisfatória com os métodos *offline*. Um dos motivos pelos quais os métodos *offline* facilitam a auto-aplicação é a divisão do 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). Na Seção 3.5 discutiremos a auto-aplicação com mais detalhe.

Mesmo após ter passado o entusiasmo inicial produzido pelos bons resultados de auto-aplicação de avaliadores parciais *offline*, essa abordagem tem se mostrado bastante eficiente na manipulação de características complexas de diversas linguagens de programação.

3.3.6 Tópicos mais Avançados de Métodos *Offline*

Assegurando a Terminação

No exemplo desenvolvido na Seção 3.3.4, utilizamos um algoritmo simples para produzir uma divisão das variáveis em estáticas e dinâmicas, na fase de BTA. Uma variável é classificada com dinâmica quando depende de um valor dinâmico em algum ponto do programa, o que convencionamos chamar de *princípio da congruência*. Como veremos a seguir, essa estratégia simples pode levar o avaliador parcial a um *loop* infinito.

Observe o trecho de programa FCL a seguir, adaptado de [55]:

```
b0:  if y ≠ 0 goto b1 else b2
b1:  x := x + 1
     y := y - 1
     goto b0
b2:  ...
```

Suponha que y seja classificada como dinâmica. Se x não é atualizada com um valor dinâmico em nenhum outro ponto do programa, a nossa estratégia simples iria classificar x como estática.

Por simplicidade, vamos supor que a única variável estática do programa seja x. Vamos supor também que, em algum momento da especialização, o bloco rotulado por $b0$ é atingido com x valendo 0. O seguinte código residual seria gerado:

```
(b0, [x ↦ 0]):  if y ≠ 0 goto (b1, [x ↦ 0]) else (b2, [x ↦ 0])
(b1, [x ↦ 0]):  y := y - 1
                goto (b0, [x ↦ 1])
(b2, [x ↦ 0]):  ...
```

Podemos observar que um novo estado foi gerado: $(b0, [x \mapsto 1])$. O avaliador deve gerar um bloco especializado para esse estado:

```
(b0, [x ↦ 1]):  if y ≠ 0 goto (b1, [x ↦ 1]) else (b2, [x ↦ 1])
(b1, [x ↦ 1]):  y := y - 1
                goto (b0, [x ↦ 2])
(b2, [x ↦ 1]):  ...
```

O processo continuaria indefinidamente. O conjunto de estados alcançáveis seria infinito. O problema é que, embora x dependa apenas de constantes e de si mesmo, o conjunto

de valores que pode receber é ilimitado, pois os valores são computados sob um controle dinâmico. Uma solução é fazer a BTA classificar todas as variáveis calculadas sob controle dinâmico como dinâmicas.

O processo de se classificar uma variável como dinâmica, mesmo quando o princípio da conguência permite que seja estática, é chamado de *generalização*. Esquemas para resolver esse problema podem ser encontrados em [53, 79, 49].

BTA com Divisão *Pointwise*

Até o momento, consideramos que uma divisão computada pela BTA é uniforme, isto é, vale para o programa inteiro. Para programas mais extensos, divisões mais especializadas podem ser necessárias para se obter melhores resultados na especialização.

Observe o seguinte trecho de código FCL:

```
b0:  x := x + 1
     y := y - 1
     goto b1
b1:  y := 0
     goto b2
b2:  ...
```

Suponha que a divisão inicial das variáveis seja (S, D), correspondendo ao par (x, y). Isto é, x é estática e y é dinâmica. Uma divisão uniforme e congruente para o programa, supondo que x não é atualizado com valores dinâmicos, seria (S, D).

Levando em conta apenas o trecho exibido, podemos propor uma divisão mais específica: `b0:`(S, D), `b1:`(S, D) e `b2:`(S, S). Uma divisão como essa é denominada como *pointwise*. Cada ponto do programa pode possuir uma divisão diferente.

Para calcular uma divisão *pointwise*, a fase de BTA tem que executar uma análise de fluxo do programa. O algoritmo de especialização que apresentamos nas seções anteriores também deveria sofrer pequenas modificações para lidar com essa nova característica.

BTA com Divisão Polivariante

Algumas vezes, até mesmo divisões *pointwise* podem ser consideradas muito restritivas. Isso pode acontecer nos casos em que a classificação de uma variável em estática ou dinâmica dependa não apenas do ponto de programa, mas da maneira como ele é alcançado.

Assumindo novamente uma divisão inicial (S, D) para o par de variáveis (x, y), observe o trecho de programa FCL a seguir:

```
b0:  if y > 0 goto b1 else b2
b1:  x := y
     goto b2
b2:  x := x + 1
     ...
```

Uma divisão pointwise congruente teria obrigatoriamente que classificar x como dinâmica nos pontos que seguem $b1$ no fluxo de controle. Assim, teria `b2:`(D, D). Podemos verificar que, se $b2$ for atingido a partir de $b0$, x poderia ser tratado como estática. Uma

divisão *polivariante* consegue lidar com essa abordagem, associando a cada rótulo um conjunto de divisões: `b0:`$\{(S, D)\}$, `b1:`$\{(S, D)\}$ e `b2:`$\{(S, D), (S, S)\}$.

3.4 Exemplos

Nesta seção mostraremos alguns exemplos de avaliação parcial de programas simples. Os programas serão codificados utilizando a linguagem FCL e o método de especialização vai empregar técnicas *offline*.

3.4.1 Casamento de Padrões

Um programa para realizar casamento de padrões em textos tem duas entradas, ambas consisitindo de uma seqüência de caracteres:

- p é o padrão procurado, tem tamanho M e é indexado de 0 a $M - 1$;
- A é o texto onde o padrão será pesquisado, tem tamanho N e é indexado de 0 a $N - 1$.

O programa retorna -1 se não encontrou nenhuma ocorrência de p em A. Caso contrário, retorna a posição da primeira ocorrência encontrada. Um algoritmo ingênuo, codificado em FCL, é apresentado a seguir:

```
(p, M, A, N)
(b0)
b0:  i := 0
     goto b1
b1:  j := 0
     goto b2
b2:  if j ≥ M goto b7 else b3
b3:  if i ≥ N goto b8 else b4
b4:  if p[j] = A[i] goto b5 else b6
b5:  i := i + 1
     j := j + 1
     goto b2
b6:  i := i - j + 1
     goto b1
b7:  return i - M
b8:  return -1
```

A tupla (p, M, A, N) indica que a entrada é formada, respectivamente, pelo padrão e seu comprimento, e pelo texto e seu comprimento. A variável j indica o caractere corrente do padrão e i indica o caractere corrente do texto. O algoritmo é ingênuo porque, se $p[j] \neq A[i]$, j passa a indicar novamente o primeiro caractere de p e i avança apenas uma posição no texto, a partir do início do último prefixo correto.

Vamos nos concentrar agora na especialização do programa apresentado em relação a um padrão específico. Ou seja, vamos produzir um novo programa, mais eficiente, que realiza a busca de um padrão específico em qualquer texto. As entradas p e M seriam, portanto classificadas como estáticas, enquanto que A e N seriam dinâmicas. Usando um método *offline*, a primeira providência é executar uma análise de tempo de definição. A divisão resultante da BTA seria:

$$\Delta = [p \mapsto S,\ M \mapsto S,\ A \mapsto D,\ N \mapsto D,\ j \mapsto S,\ i \mapsto D]\,.$$

Observe que as variáveis i e j são atualizadas apenas com resultados de operações envolvendo constantes e o valor de si próprias. Na Seção 3.3.6, quando discutimos tópicos relacionados à terminação do processo de avaliação parcial, mostramos que uma divisão congruente pode levar o processo a um *loop* infinito. No exemplo desta seção, uma análise mais elaborada mostra que j tem crescimento limitado por N, um valor conhecido, enquanto que i tem crescimento limitado por N, um valor desconhecido. Assim, é necessária uma generalização da variável i, ou seja, classificá-la como dinâmica, embora o princípio da congruência permita que seja estática.

O programa anotado, baseado na divisão calculada, é apresentado a seguir:

```
(p, M, A, N)
(b0)
b0:  i := 0
     goto b1
b1:  j := 0
     goto b2
b2:  if j ≥ M goto b7 else b3
b3:  if i ≥ N goto b8 else b4
b4:  if p[j] = A[i] goto b5 else b6
b5:  i := i + 1
     j := j + 1
     goto b2
b6:  i := i - (j - 1)
     goto b1
b7:  return i - M
b8:  return -1
```

A forma de apresentação do programa anotado é a mesma adotada na Seção 3.3.4, com as operações dinâmicas sublinhadas. Os comandos `goto` são todos sublinhados, uma vez que estamos considerando, por questões didáticas, uma fase separada para compressão de transições. As variáveis não são anotadas porque sua classificação pode ser obtida diretamente da divisão. As que aparecem sublinhadas no programa acima são devido a uma operação de *lift* implícita, ou seja, são valores estáticos que aparecem em contextos dinâmicos e são então transformados de modo a aparecer no código residual. No bloco `b6`, aplicamos uma otimização que altera a ordem das operações: a operação `(j - 1)` é realizada antes da subtração, pois envolve dois operandos estáticos e assim pode ser completamente computada.

Vamos supor agora que o programa anotado será especializado em relação aos seguintes valores: $p =$ "abc" e $M = 3$. O estado inicial será $(b0, [\text{"}abc\text{"}, 3, 0])$, indicando o bloco `b0` como ponto de programa e os valores das variáveis p, M e j, respectivamente. Uma otimização bastante simples, na geração dos estados alcançáveis, é a seguinte: se uma variável estática não é modificada em nenhum ponto do programa, ela aparecerá com o mesmo valor em todos os estados, assim pode ser eliminada da representação dos estados sem prejuízo do processo. No nosso caso, p e M nunca são alteradas, assim podemos simplificar a representação dos estados, usando apenas a variável j.

Discutiremos a seguir a geração dos estados alcançáveis e a especialização dos pontos de programa, usando como estado inicial $(b0, [0])$, onde 0 é o valor inicial da variável j.

Inicialmente, o bloco

$$(b0, [0]): \quad \texttt{i := 0}$$
$$\texttt{goto } (b1, [0])$$

é gerado, não envolvendo nenhuma computação de valores estáticos. Lembrando do algoritmo apresentado nas Figuras 3.5 e 3.9, o estado $(b1, [0])$ é acrescentado a `pending`, o conjunto dos estados ainda não gerados. Assim, o bloco

$$(b1, [0]): \quad \texttt{goto } (b2, [0])$$

é gerado em seguida. O terceiro bloco será

$$(b2, [0]): \quad \texttt{goto } (b3, [0])$$

uma vez que a condição do desvio é estática e computada como falsa. O quarto bloco configura uma situação onde a condição de um desvio é dinâmica:

$$(b3, [0]): \quad \texttt{if i} \geq \texttt{N goto } (b8, [0]) \texttt{ else } (b4, [0])$$

É a primeira vez que apresentamos uma situação como essa. Nesse caso, o conjunto `pending` é acrescido de dois novos estados, cuja ordem de processamento é irrelevante. O bloco identificado por $(b8, [0])$ gera o seguinte código, sem acrescentar novos estados:

$$(b8, [0]): \quad \texttt{return -1}$$

O bloco identificado por $(b4, [0])$ produz

$$(b4, [0]): \quad \texttt{if 'a' = A[i] goto } (b5, [0]) \texttt{ else } (b6, [0])$$

e dois novos estados: $(b5, [0])$ e $(b6, [0])$. A primeira vez que o valor de j é modificado é no bloco $(b5, [0])$:

$$(b5, [0]): \quad \texttt{i := i + 1}$$
$$\texttt{goto } (b2, [1])$$

O bloco $(b6, [0])$ nos apresenta o primeiro caso de loop produzido no código residual:

$$(b6, [0]): \quad \texttt{i := i - (-1)}$$
$$\texttt{goto } (b1, [0])$$

O bloco $(b1, [0])$ já foi gerado, assim não é acrecentado em `pending`. O conjunto `marked` é utilizado pelo algoritmo para determinar os blocos cujo código já foi gerado.

A geração dos demais blocos segue um processo similar ao descrito acima. Apresentamos em seguida o código residual, após a compressão das transições.

```
(A, N)
((b0, [0]))
(b0, [0]):  i := 0
            goto (b3, [0])
(b3, [0]):  if i ≥ N goto (b8, [0]) else (b4, [0])
(b4, [0]):  if 'a' = A[i] goto (b5, [0]) else (b6, [0])
(b5, [0]):  i := i + 1
            goto (b2, [1])
(b6, [0]):  i := i - (-1)
            goto (b3, [0])
(b8, [0]):  return -1
(b2, [1]):  if i ≥ N goto (b8, [0]) else (b4, [1])
(b4, [1]):  if 'b' = A[i] goto (b5, [1]) else (b6, [1])
(b5, [1]):  i := i + 1
            goto (b2, [2])
(b6, [1]):  i := i - (0)
            goto (b3, [0])
(b2, [2]):  if i ≥ N goto (b8, [0]) else (b4, [2])
(b4, [2]):  if 'c' = A[i] goto (b5, [2]) else (b6, [2])
(b5, [2]):  i := i + 1
            return i - 3
(b6, [2]):  i := i - (1)
            goto (b3, [0])
```

O programa residual age como um autômato, mudando de estado a cada vez que um dos caracteres do padrão é encontrado no texto, em seqüência. Se houver falha no casamento, retorna ao estado inicial, neste caso, representado por $(b3, [0])$. Uma otimização que não havia sido ainda comentada foi conduzida no código: os estados $(b8, [0])$, $(b8, [1])$ e $(b8, [2])$ foram fundidos em um único, rotulado como $(b8, [0])$, pois o código gerado para os três é idêntico.

3.4.2 Casamento de Padrões - Segunda Versão

O programa FCL para realizar casamento de padrão apresentado na seção anterior tem o seguinte inconveniente: a variável i, que indica o caractere corrente do texto, pode sofrer incrementos e decrementos. Supondo que o texto possa ser lido de um arquivo de entrada, é interessante que a variável i sofra apenas incrementos, de modo que o acesso aos caracteres possa ser realizado por uma leitura seqüencial.

Apresentamos em seguida uma nova versão para o programa de casamento de padrões, onde o texto A é lido apenas de maneira seqüencial. Ou seja, a variável i sofre apenas

incrementos. Para conseguir isso, vamos utilizar a seguinte informação:

> Se o texto A é da forma $A_0A_1 \ldots A_{N-1}$,
> e o caractere corrente do texto é indicado por i,
> e o padrão p é da forma $p_0p_1 \ldots p_{M-1}$,
> e o casamento falhou na posição j de p, com $0 \leq j \leq M-1$,
> então $A_{i-j}A_{i-j+1} \ldots A_{i-1} = p_0p_1 \ldots p_{j-1}$.

Isso significa que podemos aproveitar a informação do prefixo casado até então para determinar os caracteres do texto anteriores à posição i. É importante ressaltar que essa alteração não diminui a complexidade do algoritmo nem o torna menos ingênuo, apenas evita retroceder no texto.

O novo código é apresentado a seguir.

```
(p, M, A, N)
(b00)
b00:  i := 0
      j := 0
      goto b01
b01:  if j ≥ M goto b13 else b02
b02:  if i ≥ N goto b14 else b03
b03:  if p[j] = A[i] goto b04 else b05
b04:  i := i + 1
      j := j + 1
      goto b01
b05:  if j = 0 goto b06 else b07
b06:  i := i + 1
      goto b01
b07:  k1 := 1
      k2 := j
      j := 0
      goto b08
b08:  if k1 ≥ k2 goto b12 else b09
b09:  if p[j] = p[k1] goto b10 else b11
b10:  k1 := k1 + 1
      j := j + 1
      goto b08
b11:  k1 = k1 - j + 1
      goto b08
b12:  k1 := 0
      k2 := 0
      goto b01
b13:  return i - M
b14:  return -1
```

Boa parte do código é idêntica à primeira versão do programa para casamento de

padrões: os blocos `b00` a `b04`, `b13` e `b14`. Os blocos `b05` e `b06` tratam o caso especial que consiste na falha de casamento do primeiro caractere do padrão.

Os blocos `b07` a `b12` implementam a melhoria proposta: quando o bloco `b07` é atingido, os últimos $j-1$ caracteres do texto são exatamente o prefixo de tamanho $j-1$ do padrão. Esse tamanho é armazenado na variável $k2$. A variável $k1$ percorre o prefixo do padrão, fazendo um papel similar ao que a variável i faz com o texto. Quando o prefixo se esgota, o código é desviado para o bloco `b12` e depois a leitura do texto é reiniciada. Observe que, no bloco `b12`, a atribuição do valor 0 às variáveis $k1$ e $k2$ parece ser inútil. Veremos mais adiante que essa atribuição pode ser muito conveniente.

Como na seção anterior, vamos especializar o programa em relação a um padrão específico. Novamente, as entradas p e M são estáticas e A e N são dinâmicas. A BTA constrói a seguinte divisão:

$$\Delta = [p \mapsto S,\ M \mapsto S,\ A \mapsto D,\ N \mapsto D,\ j \mapsto S,\ i \mapsto D,\ k1 \mapsto S,\ k2 \mapsto S]\,.$$

As variáveis i e j são classificadas como estática e dinâmica, devido aos problemas relacionados com terminação discutidos na seção anterior. A variável $k2$ só é atualizada com valor constante ou com j, logo é classificada como estática. A variável $k1$ é atualizada por operações envolvendo valores constantes, o valor de j e o seu próprio valor. Como seu crescimento é limitado por $k2$, que foi determinada como estática, $k1$ também será estática.

Um trecho do programa anotado é exibido abaixo, correspondendo aos comandos que não aparecem na primeira versão do programa para casamento de padrões. Observe que nenhuma operação dos blocos `b07` a `b12` é marcada para ser residualizada.

```
      ...
b05:  if j = 0 goto b06 else b07
b06:  i := i + 1
      goto b01
b07:  k1 := 1
      k2 := j
      j := 0
      goto b08
b08:  if k1 ≥ k2 goto b12 else b09
b09:  if p[j] = p[k1] goto b10 else b11
b10:  k1 := k1 + 1
      j := j + 1
      goto b08
b11:  k1 = k1 - j + 1
      goto b08
b12:  k1 := 0
      k2 := 0
      goto b01
      ...
```

Para efeito de comparação, vamos supor novamente que o programa anotado vai ser especializado para os valores de $p =$ "*abc*" e $M = 3$, como na seção anterior. O estado inicial

será $(b00, [\text{“}abc\text{”}, 3, 0, 0, 0])$, onde os três últimos valores da lista de valores das variáveis correspondem às variáveis j, $k1$ e $k2$, respectivamente. Como p e M não sofrem alterações, vamos simplificar a representação dos estados desconsiderando essas variáveis.

Vamos discutir apenas uma parte da geração dos estados alcançáveis e especialização dos pontos de programa. Vamos nos concentrar no ponto em que o texto casou com os dois primeiros caracteres do padrão (“ab”), mas falhou no terceiro. O estado em questão é $(b05, [2, 0, 0])$, gerado por $(b03, [2, 0, 0])$:

$(b05, [2, 0, 0])$ **:** A variável j vale 2, logo é gerado um desvio para $(b07, [2, 0, 0])$.

$(b07, [2, 0, 0])$ **:** $k1, k2, j$ são atualizadas e é gerado um desvio para $(b08, [0, 1, 2])$.

$(b08, [0, 1, 2])$ **:** $k1 = 1$ e $k2 = 2$, assim é gerado um desvio para $(b09, [0, 1, 2])$.

$(b09, [0, 1, 2])$ **:** $p[j] = p[0] =$ `'a'` $\neq$ `'b'` $= p[1] = p[k1]$, assim é gerado um desvio para $(b11, [0, 1, 2])$. Note que, neste ponto, o algoritmo “percebe” que não faz sentido deslocar o padrão uma posição à direita no texto.

$(b11, [0, 1, 2])$ **:** $k1$ é atualizada e é gerado um desvio para $(b08, [0, 2, 2])$.

$(b08, [0, 2, 2])$ **:** A condição é verdadeira, gerando um desvio para $(b12, [0, 2, 2])$.

$(b12, [0, 2, 2])$ **:** k1 e k2 voltam a ser 0 e é gerado um desvio para $(b01, [0, 0, 0])$, estado que certamente já foi gerado anteriormente.

Pode-se ver que nenhum código residual foi gerado, exceto uma seqüência de desvios incondicionais que serão todos eliminados com a compressão de transições. O código residual em $(b03, [2, 0, 0])$, ponto onde o terceiro caractere do padrão é comparado com o texto, terá um formato como o seguinte:.

$(b03, [2, 0, 0])$: `if 'c' = A[i] goto` $(b04, [2, 0, 0])$ `else` $(b01, [0, 0, 0])$.

Isso significa que, se o caractere em `A[i]` não for `'c'`, o padrão começará a ser comparado do início novamente, mas o caractere corrente do texto continua o mesmo. Isto é, o padrão é deslocado duas posições à direita no texto.

O resultado acima leva a uma conclusão até certo ponto surpreendente. O programa original se baseou em um algoritmo ingênuo, *força-bruta*. Entretanto, o programa residual tem eficiência equivalente ao método KMP (Knuth-Morris-Pratt) de casamento de padrões, para um padrão específico. Esse resultado foi apresentado pela primeira vez em [23], onde os autores utilizaram um avaliador parcial para uma linguagem funcional.

Para finalizar este exemplo, vamos discutir a conveniência de se atribuir o valor 0 às variáveis $k1$ e $k2$ no bloco `b12`. Se isso não fosse feito, o estado $(b12, [0, 2, 2])$ geraria um desvio para $(b1, [0, 2, 2])$. O bloco correspondente a $(b1, [0, 2, 2])$ deveria ser então gerado, juntamente com uma longa seqüência de outros blocos. Entretanto, o código de $(b1, [0, 2, 2])$ é equivalente ao do bloco $(b1, [0, 0, 0])$. Isso acontece porque as variáveis $k1$ e $k2$ são definidas e utilizadas em um trecho do programa, mas estão “mortas” em outros trechos. Uma forma geral de se resolver esse problema é fazer uma análise de variáveis vivas e mortas, especializando cada bloco somente em relação às variáveis vivas no mesmo.

A tarefa de identificar as variáveis vivas de um bloco envolve análise de fluxo de controle do programa [1, 72]. A atribuição do valor 0 às variáveis, no exemplo, serviu como um truque para evitar a geração desnecessária de estados, supondo que a análise de variáveis vivas não está disponível.

3.4.3 Interpretador para Máquina de Turing

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.

Um estado é caracterizado por:

- um valor indicando a próxima instrução I_i, que será executada no próximo passo;
- uma fita infinita cujas células podem armazenar elementos do conjunto $\{0, 1, B\}$, onde B é "branco";
- 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 B 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 B, 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. A máquina pára quando o fluxo é desviado para um rótulo inexistente.

O programa exemplo apresentado a seguir, extraído de [55], pode provocar uma alteração na fita ou entrar em *loop* infinito, se a fita não contiver nenhum símbolo 0. Vamos chamar esse programa de MT1:

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

Vamos apresentar em seguida um interpretador para a Máquina de Turing discutida, escrito em FCL.

Para representar a fita e a cabeça de gravação, vamos utilizar duas listas: `esq` e `dir`. Suponha que o primeiro símbolo da fita diferente de B seja a_0, o último diferente de B seja a_k e que a seqüência de símbolos entre eles seja $a_0 a_1 \ldots a_k$. Se a cabeça da fita indicar a célula $a_i, 0 \leq i \leq k$, então a lista `esq` será $a_{i-1} a_{i-2} \ldots a_0$ e `dir` será $a_i a_{i+1} \ldots a_k$.

Para representar o programa da máquina, vamos utilizar uma lista `prog` de instruções, armazenadas com seus rótulos. O programa exemplo MT1 teria o seguinte formato:

[
  [0 “$ifgoto$” 0 3]
  [1 “$right$”]
  [2 “$goto$” 0]
  [3 “$write$” 1]
]

A variável `progrest` vai representar a próxima instrução I a ser executada, armazenando a lista de instruções a partir de I.

O interpretador tem como entrada a tupla (`prog`, `dir`). Como explicado acima, `prog` armazena o programa da Máquina de Turing e `dir` é a configuração inicial da fita, supondo que a cabeça de leitura/gravação indique o primeiro símbolo diferente de B. A execução do interpretador termina quando não há mais instruções para serem avaliadas, retornando o valor de `dir` no momento.

O código FCL é exibido abaixo. A função `proxinstr(`r`,prog)` foi introduzida para simplificar a especificação, e retorna uma lista de instruções de `prog`, começando pelo rótulo r. A função `hd(`L`)` retorna o primeiro elemento da lista L, `tl(`L`)` retorna o resto da lista L, descartando o primeiro elemento e `cons(`e`,`L`)` retorna uma nova lista onde e é o primeiro elemento.

```
(prog, dir)
(init)
init:     progrest := prog
          esq := []
          goto loop
loop:     if progrest = [] goto stop else cont
cont:     instr := hd (progrest)
          progrest := tl (progrest)
          op := hd (tl (instr))
          if op = ''right'' goto do-right else cont1
cont1:    if op = ''left'' goto do-left else cont2
cont2:    if op = ''write'' goto do-write else cont3
cont3:    if op = ''goto'' goto do-goto else cont4
cont4:    if op = ''ifgoto'' goto do-if else erro
do-right:  esq := cons (hd(dir), esq)
           dir := tl (dir)
           goto loop
do-left:  dir := cons (hd(esq), dir)
          esq := tl (esq)
          goto loop
do-write:  simb := hd (tl (tl(instr)))
           dir := cons (simb, tl(dir))
           goto loop
```

```
do-goto:  prox := hd (tl (tl(instr)))
          goto jump
do-if:    simb := hd (tl (tl(instr)))
          prox := hd (tl (tl (tl(instr))))
          if simb = hd(dir) goto jump else loop
jump:     progrest := proxinstr (prox, prog)
          goto loop
erro:     return 'erro de sintaxe'
stop:     return dir
```

O interpretador tem duas entradas: `prog` e `dir`. Vamos supor que o interpretador será parcialmente avaliado em relação a `prog`, isto é, um programa MT específico. A análise de tempo de definição produz a seguinte divisão:

$$\Delta = \left[\begin{array}{c} prog \mapsto S,\ dir \mapsto D,\ progrest \mapsto S,\ esq \mapsto D, \\ instr \mapsto S,\ op \mapsto S,\ simb \mapsto S,\ prox \mapsto S \end{array} \right].$$

O programa anotado associado a Δ é exibido abaixo:

```
(prog, dir)
(init)
init:     progrest := prog
          esq := []
          goto loop
loop:     if progrest = [] goto stop else cont
cont:     instr := hd (progrest)
          progrest := tl (progrest)
          op := hd (tl (instr))
          if op = ''right'' goto do-right else cont1
cont1:    if op = ''left'' goto do-left else cont2
cont2:    if op = ''write'' goto do-write else cont3
cont3:    if op = ''goto'' goto do-goto else cont4
cont4:    if op = ''ifgoto'' goto do-if else erro
do-right:  esq := cons (hd(dir), esq)
           dir := tl (dir)
           goto loop
do-left:  dir := cons (hd(esq), dir)
          esq := tl (esq)
          goto loop
do-write:  simb := hd (tl (tl(instr)))
           dir := cons (simb, tl(dir))
           goto loop
do-goto:  prox := hd (tl (tl(instr)))
          goto jump
```

```
do-if:    simb := hd (tl (tl(instr)))
          prox := hd (tl (tl (tl(instr))))
          if simb = hd(dir) goto jump else loop
jump:     progrest := proxinstr (prox, prog)
          goto loop
erro:     return 'erro de sintaxe'
stop:     return dir
```

Vamos supor agora que o programa anotado será especializado com o valor $prog = MT1$, ou seja, o programa MT dado como exemplo nesta seção.

Para discutir a geração dos estados alcançáveis e especialização dos pontos de programa, vamos supor que o avaliador parcial realiza uma análise de variáveis vivas. Um avaliador mais poderoso vai facilitar a nossa representação dos estados: vamos representar apenas os valores das variáveis vivas e que são utilizadas a partir do ponto de programa associado. Além disso, o valor da variável `progrest` será representado por um número inteiro, ao invés de uma lista, para economia de espaço: se `progrest` se refere ao programa MT que começa no rótulo r, seu valor será representado por $progrest \mapsto r$.

Para exemplificar nossa notação simplificada, observe o ponto de programa jump no código do interpretador. As únicas variáveis vivas, além de `prog` (que não sofre alteração), são `progrest` e `prox`. Assim, um estado associado a esse ponto de programa será representado por $(jump, [progrest \mapsto \ldots, prox \mapsto \ldots])$. Se `progrest` indicar o programa a partir da instrução 1, a representação será $(jump, [progrest \mapsto 1, prox \mapsto \ldots])$. Se `progrest` for a lista vazia, será representado por um rótulo não existente no programa MT.

O estado inicial é $(init, [\ ])$, produzindo o código

```
(init, [ ]):  esq := []
              goto (loop, [progrest ↦ 0])
```

no programa residual. No estado $(loop, [progrest \mapsto 0])$, uma série de desvios é gerada, sem nenhum código extra, até atingir o ponto onde a variável `op` é verificada ser igual a "$ifgoto$":

```
(cont4, [progrest ↦ 1, instr ↦ [0 "ifgoto" 0 3], op ↦ "ifgoto"]):
    goto (do-if, [progrest ↦ 1, instr ↦ [0 "ifgoto" 0 3]])
```

No processamento do novo estado gerado, o seguinte código é produzido:

```
(do-if, [progrest ↦ 1, instr ↦ [0 "ifgoto" 0 3]]):
    if 0 = hd(dir) goto
        (jump, [progrest ↦ 1, prox ↦ 3])
    else
        (loop, [progrest ↦ 1])
```

Agora temos dois novos estados para processar. O primeiro produz o código

```
(jump, [progrest ↦ 1, prox ↦ 3]):
    goto (loop, [progrest ↦ 3])
```

assim teremos dois estados no conjunto `pending` associados ao ponto de programa `loop`, $(loop, [progrest \mapsto 3])$ e $(loop, [progrest \mapsto 1])$, indicando o processsamento a partir dos comandos rotulados por 1 e 3, do programa `prog`.

Vamos nos concentrar primeiro em $(loop, [progrest \mapsto 3])$. Novamente, uma série de desvios é gerada, sem nenhum código extra, até atingir o ponto onde a variável `op` é verificada ser igual a "*write*". Então a seqüência de código a seguir é produzida:

$(cont2,\ [progrest \mapsto 4, instr \mapsto [3\ \text{“}write\text{”}\ 1], op \mapsto \text{“}write\text{”}])$:
    `goto` $(do\text{-}write,\ [progrest \mapsto 4, instr \mapsto [3\ \text{“}write\text{”}\ 1]])$
$(do\text{-}write,\ [progrest \mapsto 4, instr \mapsto [3\ \text{“}write\text{”}\ 1]])$:
```
    dir := cons (1, tl(dir))
    goto (loop, [progrest ↦ 4])
```
$(loop,\ [progrest \mapsto 4])$:
    `goto` $(stop,\ [\,])$
$(stop,\ [\,])$:
    `return dir`

Para o estado $(loop, [progrest \mapsto 1])$, a seguinte seqüência de código será produzida (simplificada com algumas compressões de transição):

$(loop,\ [progrest \mapsto 1])$:
    `goto` $(do\text{-}right,\ [progrest \mapsto 2, instr \mapsto [1\ \text{“}right\text{”}]])$
$(do\text{-}right,\ [progrest \mapsto 2, instr \mapsto [1\ \text{“}right\text{”}]])$:
```
    esq := cons (hd(dir), esq)
    dir := tl (dir)
    goto (loop, [progrest ↦ 2])
```
$(loop,\ [progrest \mapsto 2])$:
    `goto` $(do\text{-}goto,\ [progrest \mapsto 3, instr \mapsto [2\ \text{“}goto\text{”}\ 0]])$
$(do\text{-}goto,\ [progrest \mapsto 3, instr \mapsto [2\ \text{“}goto\text{”}\ 0]])$:
    `goto` $(jump,\ [progrest \mapsto 3, prox \mapsto 0]])$
$(jump,\ [progrest \mapsto 3, prox \mapsto 0]])$:
    `goto` $(loop,\ [progrest \mapsto 0])$

O estado $(loop, [progrest \mapsto 0])$ já foi gerado, dando origem então a um *loop* no programa residual. Como todos os estados foram processados, a especialização dos pontos de programa termina.

O programa residual final, após compressão de transições e renomeação de rótulos, é apresentado a seguir:

```
(dir)
(b0)
b0:    esq := []
       goto b1
b1:    if 0 = hd(dir) goto b2 else b3
b2:    dir := cons (1, tl(dir))
       return dir
b3:    esq := cons (hd(dir), esq)
       dir := tl (dir)
       goto b1
```

Uma situação interessante pode ser observada no resultado obtido acima. O programa residual tem exatamente a mesma semântica que o programa MT1. A diferença é que o primeiro está escrito em FCL e o segundo, na linguagem da Máquina de Turing.

Vejamos como isso aconteceu:

1. O avaliador parcial recebe um programa FCL e parte de sua entrada, gerando um novo programa FCL que, quando aplicado ao restante da entrada, produz os mesmos resultados que original, se aplicado à entrada completa.

2. O programa a ser parcialmente avaliado é um interpretador para a linguagem da Máquina de Turing.

3. O interpretador recebe como dados de entrada um programa MT e o estado inicial da fita.

4. A execução do interpretador sobre um programa MT e uma fita retorna o novo estado da fita, ou seja, o interpretador simula a semântica da execução do programa MT sobre uma fita inicial.

5. O interpretador é parcialmente avaliado em relação a um programa específico MT1.

6. O resultado é um programa FCL que, quando aplicado à fita de entrada, produz o mesmo resultado que a execução do interpretador sobre MT1 e a fita inicial, que é o mesmo resultado da execução de MT1 sobre a fita.

7. Conclusão: o programa residual é o resultado da **compilação** de MT1, escrito na linguagem da Máquina de Turing, para a linguagem FCL.

Na Seção 3.5 veremos uma generalização e extensões dessa conclusão.

3.5 Geração de Geradores de Programas

Nesta seção, vamos dar um tratamento um pouco mais formal à avaliação parcial de programas. Mostraremos como pode ser utilizada para compilação e geração de compiladores dirigida por semântica. Finalmente, discutimos uma abordagem diferente para geração de geradores de compiladores.

Seguindo a notação utilizada em [55], 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:

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

O programa P, quando aplicado às entradas, in_1 e in_2, produz a saída out. O avaliador parcial `mix`, quando aplicado a P e parte de sua entrada (in_1), produz um novo programa

identificado como P_{in_1}. O programa residual P_{in_1} produz a mesma saída *out*, quando a entrada restante (in_2) é submetida a ele. 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.

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. Nos exemplos apresentados na Seção 3.3, as linguagens S e T são a linguagem de fluxogramas FCL. Não discutimos em qual linguagem o próprio avaliador parcial estaria implementado, embora tenhamos apresentado um código para mix na Figura 3.5, usando uma linguagem algorítimica. Com pouco esforço, os algoritmos das Figuras 3.5, 3.6 e 3.9 podem ser traduzidos para a linguagem FCL.

3.5.1 Compilação e Geração de Compiladores

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, e as equções que a descrevem são conhecidas como as *três projeções de Futamura* [33].

Supondo *int* um interpretador de uma linguagem qualquer, escrito em S, a primeira projeção de Futamura mostra que compilação por meio 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 em relação a um programa fonte específico. Pudemos observar a aplicação desse procedimento no exemplo da Seção 3.4.3, quando realizamos compilação de um programa escrito na linguagem da Máquina de Turing para a linguagem FCL.

Um interpretador para uma linguagem L pode ser visto como uma descrição da semântica de L. Assim, a primeira projeção de Futamura mostra que é possível realizar *compilação dirigida por semântica*, usando um avaliador parcial `mix`.

O avaliador parcial `mix` é um programa que recebe duas entradas: um programa P a ser especializado e parte dos dados de entrada de P. Assim, o próprio programa `mix` pode ser especializado em relação a P.

A segunda projeção de Futamura refere-se à geração de compiladores por meio de auto-aplicação de `mix`:

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

Temos então $compiler = [\![mix]\!]\,(mix, int)$. Um compilador é gerado por meio da avaliação parcial do próprio avaliador parcial, em relação a um interpretador específico: *geração de compiladores dirigida por semântica*.

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. Nos exemplos da Seção 3.3, tanto a linguagem fonte dos programas submetidos a `mix`, quanto a linguagem dos programas residuais produzidos, eram a linguagem FCL. Para podermos aplicar os procedimentos descritos nesta seção, seria necessário codificar `mix` em FCL.

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 [57] requeria anotações prévias introduzidas pelo usuário, mas uma versão seguinte [58] era completamente automática.

Joergensen realizou experimentos que envolviam a geração de um compilador para uma linguagem funcional de avaliação *lazy*, usando um avaliador parcial escrito em uma linguagem funcional de avaliação estrita [61]. Os experimentos mostraram que a velocidade de execução do código compilado foi equivalente ao produzido por compiladores comerciais.

A auto-aplicação de `mix` pode ir ainda mais longe. 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 programa `cogen` é chamado de gerador de compiladores, porque recebe um interpretador para uma linguagem L como entrada, produzindo um compilador de L para a linguagem dos programas residuais de `mix`.

Muitos experimentos envolvendo a geração automática de geradores de compiladores, usando a auto-aplicação de um avaliador parcial, foram bem sucedidos. A maioria tinha como linguagem fonte uma linguagem não tipada [58, 36, 24, 60, 61, 48].

3.5.2 Extensões de Geração

Na realidade, o programa `cogen` apresentado na seção anterior é mais do que um gerador de compiladores. Se `cogen` for aplicado a um programa P qualquer, podendo ser um interpretador ou não, produz uma *extensão de geração (generating extension)* para P. Uma extensão de geração de um programa P é um programa P_{gen} que, quando executado com

um valor in_1 para a primeira entrada de P, gera um programa residual P_{in_1}. O programa P_{in_1} é o resultado da avaliação parcial de P com valor in_1 para a primeira entrada.

Para facilitar o entendimento, vamos apresentar um exemplo onde uma extensão de geração simples é produzida. Para isso, vamos utilizar o primeiro exemplo deste capítulo, que é a função Power escrita em C, exibida na Figura 3.1. Power possui duas entradas, denominadas n e x. Uma extensão de geração para Power é um programa que, quando recebe um valor in_1, produz uma função $Power_{in_1}$, resultado da especialização de Power em relação a $n = in_1$. Uma extensão de geração Power_gen para a função Power é exibida a seguir.

```
void Power_gen (int n) {
    printf(''int power_%d(int x)\n'', n);
    printf(''{ int p = 1;\n'');
    while (n > 0) {
        if (n%2 == 0) {
            printf(''x = x * x\n'');
            n = n / 2;
        }
        else {
            printf(''p = p * x;\n'');
            n = n - 1;
        }
    printf(''return p;\n'');
    printf(''} \n'');
}
```

Ao executar Power_gen com $n = 5$, obtemos o seguinte programa residual:

```
int Power_5 (int x) {
    int p = 1;
    p = p * x;
    x = x * x;
    x = x * x;
    p = p * x;
    return p;
}
```

Voltando a `cogen`, a idéia por trás da terceira projeção de Futamura é gerar automaticamente um gerador de extensões de geração, usando auto-aplicação de um avaliador parcial `mix`. Em especial, se `cogen` for aplicado a um interpretador, a extensão de geração produzida é na realidade um compilador. A segunda e terceira projeções de Futamura podem ser generalizadas da seguinte forma:

$$[\![mix]\!]\,(mix, P) \;=\; P_{gen} \qquad \text{(segunda projeção)}$$

$$\begin{aligned} [\![mix]\!]\,(mix, mix) &= cogen \qquad \text{(terceira projeção)} \\ [\![cogen]\!]\,(P) &= P_{gen} \end{aligned}$$

Na década de 90, uma abordagem que tornou-se popular foi a de escrever um gerador de extensões de geração `cogen` à mão [63, 4, 13], em vez de se construir um avaliador parcial `mix`. O gerador `cogen` pode ser utilizado para realizar avaliação parcial de um programa P de modo tradicional. Para isso, basta gerar uma extensão de geração para P, então aplicar essa extensão a um valor específico, produzindo um programa especializado. Por outro lado, vimos que `cogen` pode ser automaticamente gerado a partir da auto-aplicação de `mix`. Assim, as duas abordagens parecem ser equivalentes. Então por que razão construir um gerador de extensões de geração em vez de um avaliador parcial auto-aplicável? Em [70], as seguintes razões são enumeradas:

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.

2. Pela razão acima, entre outras, pode ser mais fácil escrever um gerador de extensões de geração do que um avaliador parcial auto-aplicável.

3. Um avaliador parcial deve conter um meta-interpretador, o que pode ser um problema sério para linguagens estaticamente tipadas, como será discutido a seguir. Nem o gerador de extensões de geração, nem as extensões de geração produzidas, precisam conter um meta-interpretador.

Quando se escreve um interpretador para uma linguagem estaticamente tipada, um único tipo universal deve ser utilizado no interpretador para representar um número ilimitado de tipos utilizado pelos programas que são interpretados. O mesmo é válido para um avaliador parcial auto-aplicável, pois ele contém um meta-interpretador, isto é, um interpretador da própria linguagem em que está escrito. Isso pode causar problemas de ineficiência, quando o programa residual herda as estruturas para tratamento do tipo universal.

Na primeira projeção de Futamura, temos

$$[\![mix]\!]\,(int, source) = target,$$

onde o programa residual *target* é formado por partes de *int*. Nesse caso, o problema descrito acima não é verificado.

Na segunda projeção de Futamura, temos

$$[\![mix]\!]\,(mix, int) = compiler.$$

Nesse caso, o programa residual *compiler* é formado por partes do próprio `mix`. Como `mix` utiliza um tipo universal para tratar os tipos encontrados no interpretador *int*, o compilador *compiler* herda essa ineficiência.

O problema é ainda mais sério quando aplicamos a terceira projeção:

$$[\![mix]\!]\,(mix, mix) = cogen.$$

O gerador de compiladores *cogen* tem uma execução ineficiente, e além disso, os compiladores gerados por ele também são ineficientes. O fato de conter um tipo universal para tratar

todos os tipos da linguagem faz com que um programa de uma linguagem estaticamente tipada se comporte como o de uma linguagem não tipada, perdendo assim as vantagens de eficiência das linguagens estaticamente tipadas.

Um gerador de extensões de geração escrito à mão transforma um programa escrito em uma linguagem L em outro da mesma linguagem. Assim não precisa conter um interpretador, e um compilador pode ser gerado sem auto-aplicação. As extensões de geração produzidas, bem como os programas especializados, não herdam nenhum mecanismo para tratamento de um tipo universal.

As conclusões que se pode tirar são as seguintes:

- Para linguagens não tipadas, resultados satisfatórios podem ser conseguidos na geração de compiladores dirigida por semântica, usando auto-aplicação de um avaliador parcial.

- Para linguagens estaticamente tipadas, é mais adequado construir `cogen` à mão. A avaliação parcial tradicional pode ser conduzida como descrevemos anteriormente, e é possível a geração de compiladores mais eficientes.

3.6 Leitura Adicional

O texto deste capítulo se concentrou na especialização de programas escritos na linguagem FCL. Linguagens reais ou que seguem outros paradigmas, como as linguagens funcionais, podem utilizar os princípios básicos discutidos para construir um avaliador parcial.

Nesta seção, apresentaremos uma série de textos que podem ser utilizados como fonte de informações sobre avaliação parcial de programas. Procuramos citar as referências mais importantes e que cobrem também tópicos não abordados neste capítulo.

O texto base para entendimento de avaliação parcial de programas é o livro de Jones, Gomard e Sestoft [55]. O livro cobre tópicos iniciais, como as definições básicas dos conceitos envolvidos, algoritmos para implementação de avaliadores parciais e exemplos, de forma bastante didática. Trata também aspectos relacionados à especialização de programas escritos em linguagens de paradigmas diversos: imperativo, funcional e lógico. Os capítulos finais apresentam tópicos mais avançados, como garantia de terminação da avaliação parcial, supercompilação etc.

Um segundo livro foi publicado sobre o assunto três anos depois, tendo como autores Danvy, Glück e Thiemann [25]. Na realidade, trata-se de uma reunião de artigos completos, procurando resumir o estado da arte e as perspectivas futuras da avaliação parcial de programas.

O artigo de Mogensen e Sestoft [70] é ideal para iniciantes em avaliação parcial, pois é um tutorial mais atualizado. Em poucas páginas, aborda a maioria dos aspectos mais importantes relacionados ao tema. Em particular, trata, com muito mais detalhe que o livro de Jones, Gomard e Sestoft, da geração à mão de geradores de extensões de geração. Algumas referências bibliográficas citadas são mais recentes.

A Escola de Verão de 1998 do Departamento de Ciência da Computação da Universidade de Copenhagen (DIKU) abordou o tema: *Avaliação Parcial - Teoria e Prática*. Os textos dos seminários apresentados no evento são também uma leitura muito interessante, publicados posteriormente em [47]. O material inclui os textos e transparências utilizadas por John Hatcliff, que serviram como umas das principais bases do texto apresentado neste capítulo, em especial as seções 3.3.3 e 3.3.4.

O grupo de pesquisa TOPPS, da Universidade de Copenhagen, estuda aspectos relacionados à manipulação semântica de programas e compreende boa parte dos pesquisadores citados nas referências deste capítulo. No *site* do grupo[1] pode-se obter cópias eletrônicas de muitos dos artigos citados e outras informações importantes.

3.7 Conclusões

Neste capítulo, vimos os principais conceitos relacionados à avaliação parcial de programas. Mostramos que o objetivo da avaliação parcial é produzir programas mais eficientes, se parte da entrada é conhecida. Entre as aplicações mais importantes, está a geração automática de compiladores.

Para a construção de um avaliador parcial, diferentes técnicas podem ser necessárias, dependendo da linguagem fonte dos programas que serão processados. Mais precisamente, o paradigma da linguagem fonte é um fator importante a ser considerado. O texto se concentrou em técnicas relacionadas a linguagens imperativas, adotando uma linguagem de fluxograma simples como exemplo, denominada FCL e definida na Seção 3.3.1. Entretanto, a técnica de Especialização Polivariante apresentada na Seção 3.3.2 pode ser facilmente adaptada a outros paradigmas de programação.

As técnicas para produção de um avaliador parcial tradicional foram divididas em duas abordagens: *online* e *offline*. A abordagem *online*, discutida na Seção 3.3.3, foi muito utilizada pelos primeiros avaliadores parciais. Um programa especializado é produzido em um único passo. A abordagem *offline*, por outro lado, consiste na construção de um programa anotado que indica as partes que deverão ser computadas e as partes que deverão ser residualizadas. Um segundo passo promove a especialização, seguindo estritamente essas anotações. Essa abordagem foi discutida na Seção 3.3.4.

Alguns exemplos de aplicações das técnicas de avaliação parcial foram apresentados na Seção 3.4, utilizando a abordagem *offline*. O exemplo da Seção 3.4.3 mostra a especialização de um interpretador de uma versão da Máquina de Turing, escrito em FCL. O resultado foi a compilação da linguagem da Máquina de Turing para a linguagem FCL. Mostramos como esse resultado pode ser generalizado, apresentando as equações conhecidas como as Três Projeções de Futamura, na Seção 3.5. Essas equações mostram como compilação, geração de compiladores e geração de geradores de compiladores podem ser conseguidas por meio da utilização de um avaliador parcial.

A abordagem das projeções de Futamura não é adequada a linguagens estaticamente tipadas, quando o objetivo é a geração automática de compiladores usando avaliação parcial. Uma solução mais eficiente é a aplicação de um gerador de extensões de geração.

[1] www.diku.dk/research-groups/TOPPS

Na Seção 3.5.2, apresentamos essa nova abordagem e discutimos as razões de sua maior eficiência.

As técnicas apresentadas neste capítulo são utilizadas em nossa pesquisa de duas maneiras diferentes. Primeiramente, trabalhamos com uma versão dinamicamente tipada da linguagem ASM. Construímos então um avaliador parcial tradicional para essa linguagem, usando a abordagem *offline*. A segunda parte da nossa pesquisa envolveu o desenvolvimento de um gerador de extensões de geração para ASM.

Os capítulos seguintes tratam da aplicação das técnicas de avaliação parcial discutidas neste capítulo, dentro da nossa pesquisa. No Capítulo 4 a seguir, mostramos os resultados produzidos pelo avaliador parcial *offline* construído, utilizando como linguagem fonte uma linguagem ASM dinamicamente tipada. No Capítulo 7, descrevemos detalhadamente o desenvolvimento e as experiências com um gerador de extensões de geração para ASM.

Capítulo 4

Avaliação Parcial para ASM

A proposta original de Gurevich para a linguagem das Máquinas de Estado Abstratas [40] não especifica uma verificação de tipos estática ou dinâmica. Tomando como base programas ASM não tipados, desenvolvemos um avaliador parcial para a linguagem ASM, apresentado neste capítulo.

Uma importante extensão introduzida na linguagem ASM foi a possibilidade de especificar algumas funções usando uma expressão para o cálculo de seus valores. Essa expressão pode conter chamadas recursivas da própria função, seguindo o paradigma funcional tradicional, com avaliação estrita. Essa extensão passou a ser designada por *derived functions* [26]. O avaliador parcial desenvolvido trata também esse tipo de especificação, assim foi necessário combinar técnicas de avaliação parcial de linguagens imperativas e de linguagens funcionais com as novas técnicas desenvolvidas.

O primeiro avaliador parcial construído para ASM que se tem notícia foi apresentado por Gurevich e Huggins em [43] e utilizava técnicas *offline*. Não permitia entretanto a auto-aplicação, impossibilitando verificar os resultados da aplicação da segunda e terceira projeções de Futamura, nem processava a especificação de funções definidas recursivamente (*derived functions*).

O trabalho que desenvolvemos também utiliza métodos *offline*, mas permite a geração de compiladores usando a auto-aplicação, assunto abordado na Seção 4.6. O processo é dividido, como normalmente, nas fases de análise de tempo de definição e especialização propriamente dita.

4.1 Linguagem Processada

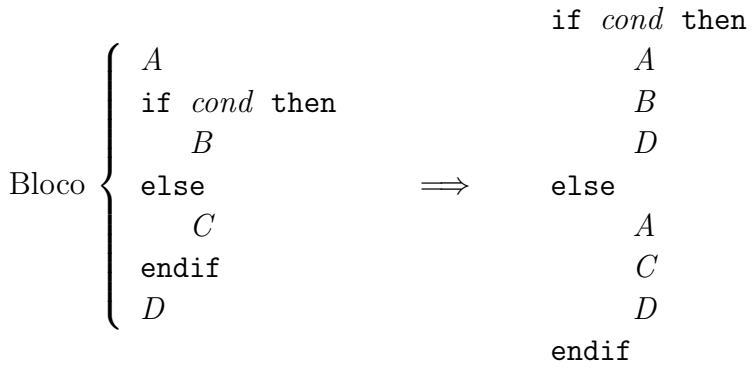
O avaliador parcial processa apenas regras básicas ASM, isto é, comandos condicionais, regras de atualização e blocos de regras.

Apesar de ser apenas um subconjunto da linguagem ASM, as regras básicas são poderosas o bastante para descrever a semântica de processos complexos. Um exemplo é a descrição da semântica da linguagem C [42]. Uma extensão para a linguagem processada é discutida na Seção 4.5.

4.2 Pré-Processamento

Huggins [43] propõe um pré-processamento do programa de modo a poder simplificar o código do avaliador parcial. Esse pré-processamento pode ser realizado antes ou depois da fase de análise de tempo de definição.

Nosso avaliador parcial executa um processo similar ao de Huggins, consistindo em eliminar a ocorrência de comandos condicionais dentro de blocos de comandos. Se um bloco contém um comando condicional e ainda outros comandos $c_1, c_2, \ldots, c_k$, o bloco é reduzido ao comando condicional, com $c_1, c_2, \ldots, c_k$ duplicados dentro das cláusulas THEN e ELSE desse comando. A figura a seguir apresenta um esquema dessa transformação, onde A, B, C e D são um conjunto qualquer de regras ASM:



Esse processo é repetido recursivamente, até que a regra de transição seja reduzida a uma árvore: os nodos internos serão comandos condicionais e as folhas serão blocos contendo apenas regras de atualização.

Essa transformação pode fazer um programa crescer exponencialmente. A descrição da semântica de uma linguagem de programação em ASM é um interpretador, tendo geralmente o formato de uma longa cadeia de comandos condicionais que atribuem semântica a cada comando da linguagem interpretada. Programas assim têm pouca chance de serem muito expandidos pela transformação descrita nesta seção.

4.3 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 no programa (pré-processado) P.

O algoritmo funciona da forma descrita a seguir. A cada iteração, uma função f será classificada como negativa se existir um termo $f(t^*)$ onde t^* é uma tupla de termos com qualquer referência a uma função negativa. Uma função f também será classificada como negativa se existir uma atualização $f(t^*) := t_0$ onde 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 [55]. 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 por meio de anotações à mão.

Depois de classificar todas as funções referenciadas em P, cada operação e comando é também classificado. Termos $f(t^*)$ e atualizações $f(t^*) := t_0$ serão negativos se f foi classificada como negativa. Um comando condicional será negativo se a condição testada for negativa. Em todos os outros casos, essas construções serão classificadas como positivas, gerando uma regra anotado designado por P_1.

4.4 Especialização

A especialização é conduzida sobre o programa P_1 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_1 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_1 é processada da seguinte maneira:

- 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.

- Se a regra for um bloco, deverá conter apenas regras de atualização, devido ao pré-processamento. Cada regra é processada como a seguir:

  – Um comando de atualização positivo gera uma atualização no estado corrente.

  – Um comando de atualização negativo $f(\bar{t}) := t_0$ produz como código uma atualização com todas as estruturas positivas de $\bar{t}$ e t_0 avaliadas.

  As atualizações acumuladas são então processadas sobre o estado corrente, em paralelo, e um (novo) estado é gerado. A cada diferente estado gerado, é atribuído um identificador único. No final do bloco residual, é gerado um código que referencia esse estado gerado. Isso irá definir a ordem de execução dos comandos no programa residual.

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 seja 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 `curstate`. O código no final de cada bloco residual será a atualização `curstate` $:= i$, onde i é o estado referenciado por esse comando, conforme estabelecido acima. O código do corpo do programa residual será uma seqüência de testes executados sobre o valor de `curstate`.

4.4.1 Exemplo

Para demonstrar o funcionamento do avaliador parcial desenvolvido, vamos utilizar o programa do Exemplo 5, do Capítulo 2. Vamos supor que esse programa será especializado em relação ao programa MT exibido no mesmo exemplo. O programa residual é apresentado na Figura 4.1, antes da aplicação da compressão das transições.

Uma série de otimizações deve ser implementada de modo a tornar o programa residual mais eficiente. Uma otimização de fluxo de controle simples é a compressão das transições, que em linguagens imperativas significaria eliminação de `gotos` redundantes. Essa situação é representada por uma regra da forma

```
IF curstate = i THEN curstate := j ENDIF
```

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

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

Figura 4.1: Exemplo de especialização do interpretador de MT.

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

Figura 4.2: Exemplo com compressão de transições.

de regras são realizadas durante o processo de especialização (*transition compression on the fly*), ao invés de serem feitas em uma 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 uma especificação como a apresentada na Figura 4.2.

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*. Esse resultado corresponde à Primeira Projeção de Futamura.

Outras experiências envolvendo compilação por meio da primeira projeção de Futamura foram realizadas. Os testes compreenderam a especialização de interpretadores de lingua-

gens mais complexas, como por exemplo um subconjunto da linguagem C. Os resultados alcançados foram satisfatórios.

4.4.2 Otimizações

Algums otimizações interessantes foram implementadas no avaliador parcial, sendo fundamentais para os bons resultados obtidos. Além da *compressão de transições*, já discutida anteriormente, vamos citar mais duas nesta seção.

Na fase de BTA, uma função f com aridade k terá seus parâmetros $p_1, \ldots, p_k$ divididos em estáticos e dinâmicos. Suponha $s_1, \ldots, s_i$ o conjunto de parâmetros estáticos, e $d_1, \ldots, d_j$ o conjunto de parâmetros dinâmicos, com $k = i + j$. No programa residual, uma função diferente será produzida para cada conjunto de diferentes valores estáticos atribuídos a $s_1, \ldots, s_i$ durante toda a fase de especialização. Essas funções residuais terão aridade j.

Outra otimização importante é uma técnica conhecida como *arity raising* [55, 77]. De modo simplificado, essa técnica funciona como descrito a seguir. Suponha que um dos parâmetros de uma função f seja uma lista com valores dinâmicos, mas cujo tamanho é conhecido, ou seja, um valor estático. A técnica de *arity raising* consiste em transformar esse único parâmetro em vários parâmetros, possivelmente aumentando a aridade da função f. Isso é particularmente útil em interpretadores, quando a descrição de uma chamada de função utiliza uma lista para agrupar seus parâmetros. A técnica faz com que os programas residuais se tornem mais parecidos com o programa sendo interpretado.

Essas e outras otimizações foram especialmente importantes para o bom resultado obtido com o teste do meta-interpretador, que é discutido na Seção 4.5 a seguir.

4.5 Teste do Meta-Interpretador

Um teste é sugerido em [55] 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`), ou seja, um interpretador para S escrito na própria linguagem S. Em seguida, o meta-interpretador deve ser avaliado 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 o mais similar possível ao fornecido como entrada para o meta-interpretador.

Para realizar esse teste, foi construído um meta-interpretador para ASM. Os resultados alcançados satisfizeram os critérios impostos pelo teste, isto é, a especialização do meta-interpretador com relação a vários programas ASM P gerou como resultado programas residuais muito similares a P, sendo a única diferença a renomeação das funções utilizadas em P. As otimizações discutidas na Seção 4.4.2 foram fundamentais para alcançar bons resultados.

Entretanto, outro aspecto foi muito importante para chegar a esses bons resultados. Sentimos a necessidade de especificar o meta-interpretador usando não apenas as regras básicas. O resultado foi atingido porque usamos também um comando equivalente ao

comando `var`, introduzido na Definição 13 do Capítulo 2. Tivemos que promover alterações no pré-processamento, bem como especificar o modo como o avaliador parcial processa o comando `var` nas fases de análise de tempo de definição e especialização.

Um problema sério foi detectado. A especialização do meta-interpretador em relação a um programa ASM P gerou como resultado um programa residual equivalente a P, com a seguinte restrição: P só poderia utilizar regras básicas. Ou seja, o meta-interpretador utiliza uma regra `var`, mais complexa, mas só interpreta programas ASM que usam blocos, comandos condicionais e regras de atualização. O meta-interpretador não poderia, por exemplo, ser especializado em relação a si próprio.

Não conseguimos especificar a interpretação da regra `var` de modo a gerar resultados satisfatórios para a especialização do meta-interpretador. Isso trouxe problemas também para a auto-aplicação do avaliador parcial, já que a especificação de um avaliador parcial passa necessariamente pela especificação de um meta-interpretador. Isto é, um avaliador parcial **contém** um meta-interpretador. Vamos discutir novamente esse assunto na seção seguinte.

4.6 Geração de Compiladores

A segunda projeção de Futamura, como visto na Seção 3.5, 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 foi implementado em Java e aplicado a programas escritos na linguagem ASM não-tipada. Vamos designar esse avaliador parcial por $\texttt{mix}_{Java}$. Para obter a geração de compiladores dirigida por semântica, uma nova versão $\texttt{mix}_{ASM}$ foi escrita na linguagem ASM, de modo a poder servir de entrada para $\texttt{mix}_{Java}$. O avaliador $\texttt{mix}_{ASM}$ não é tão poderoso quanto $\texttt{mix}_{Java}$, implementando apenas a fase de especialização do método *offline*. A simplificação de $\texttt{mix}_{Java}$ é uma característica 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 $\texttt{mix}_{ASM}$ 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 $\texttt{mix}_{ASM}$ 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 $\texttt{mix}_{Java}$.

Suponha que `int` seja um interpretador de uma linguagem L escrito em ASM. O programa $\texttt{int}_{an}$ é o resultado da adição das anotações de BTA a esse interpretador. Temos então:

$$compiler = [\![\texttt{mix}_{Java}]\!] \, (\texttt{mix}_{ASM}, \texttt{int}_{an}) \, .$$

Isto é, por meio da Segunda Projeção de Futamura, um compilador de L para a linguagem ASM é automaticamente gerado.

O compilador gerado pela aplicação da equação acima ao interpretador do Exemplo 5, do Capítulo 2, ficou bastante extenso (mais de 10 vezes o tamanho do interpretador). Em média, a compilação utilizando esse compilador foi três vezes mais rápida do que a compilação via avaliação parcial. Entretanto, o código gerado por esse compilador para programas MT simples carece de muitas otimizações, principalmente a compressão de transições.

Os resultados com geração de compiladores usando a versão dinamicamente tipada da linguagem ASM foram insatisfatórios. Isso se deveu à ausência de muitas otimizações em $\texttt{mix}_{ASM}$, o avaliador parcial escrito em ASM. Mesmo com a implementação de otimizações, os compiladores gerados serão de pouca utilidade, pois são escritos em ASM e geram código em ASM (versão dinamicamente tipada). Entretanto, a implementação de uma versão de $\texttt{mix}_{ASM}$ eficiente ainda é interessante, pois corresponde a uma descrição formal de um avaliador parcial auto-aplicável para ASM. O Capítulo 5 apresenta uma descrição detalhada de $\texttt{mix}_{ASM}$.

Como discutimos na Seção 4.5, a utilização da regra `var` foi necessária para se obter bons resultados na especialização do meta-interpretador de ASM. O avaliador parcial $\texttt{mix}_{ASM}$ contém um meta-interpretador de ASM, assim sofre os mesmos problemas que esse meta-interpretador. Na prática, isso faz com que $\texttt{mix}_{Java}$ seja necessariamente mais poderoso que $\texttt{mix}_{ASM}$, o que não configura uma "auto-aplicação" real. No Capítulo 5, voltamos a abordar esse assunto.

4.7 Conclusão

Como explicado no Capítulo 1, inicialmente concentramos nosso trabalho em uma versão dinamicamente tipada da linguagem ASM. Construímos um avaliador parcial para essa linguagem, que utiliza métodos *offline* para produzir programas especializados. A linguagem utilizada na implementação foi Java, e o avaliador parcial foi denominado $\texttt{mix}_{Java}$.

O avaliador parcial construído pode ser considerado de boa qualidade, pois os resultados do teste do meta-interpretador [55] foram bastante satisfatórios. Esse teste consistiu em construir um interpretador para ASM escrito na própria linguagem. Ao se especializar esse meta-interpretador com respeito a programas ASM específicos, os resultados produzidos foram programas praticamente equivalentes aos fornecidos como entrada. Isso mostrou que o avaliador parcial desenvolvido é poderoso o bastante para eliminar todo o *overhead* de interpretação. Esses resultados foram conseguidos para uma linguagem ASM contendo apenas comandos básicos.

Um dos principais resultados de nossa pesquisa é a geração automática de compiladores dirigida por semântica. A linguagem fonte com a qual lidamos nas experiências descritas neste capítulo é dinamicamente tipada, o que a tornaria adequada para geração de compiladores usando as projeções de Futamura. A aplicação da Primeira Projeção de Futamura obteve resultados satisfatórios, com a compilação de programas escritos em algumas linguagens simples para a linguagem ASM dinamicamente tipada. Uma das linguagens utilizadas nas experiências foi um subconjunto de C.

Para a aplicação da Segunda Projeção de Futamura, desenvolvemos uma nova versão do avaliador parcial, escrita na própria linguagem ASM, permitindo a "auto-aplicação". Esse

segundo avaliador parcial foi denominado $\texttt{mix}_{ASM}$ e é discutido em detalhes no Capítulo 5. Os compiladores gerados como resultado foram muito extensos e ineficentes. A causa do relativo insucesso da geração de compiladores foi a falta de muitas otimizações em $\texttt{mix}_{ASM}$.

Encontramos muitas dificuldades em expressar as principais otimizações do avaliador parcial na própria linguagem ASM. Isso afetou a qualidade dos compiladores gerados usando a Segunda Projeção de Futamura, mas um problema ainda mais sério ocorre se utilizarmos a Terceira Projeção de Futamura. Esse problema está relacionado à necessária utilização do comando `var` no código de $\texttt{mix}_{ASM}$, e é discutido com detalhe no Capítulo 5. Assim, em vez de adotarmos então a abordagem da Terceira Projeção de Futamura, adotamos a abordagem de extensões de geração, para produzir um gerador de geradores de compiladores eficiente para ASM. Essa experiência é apresentada no Capítulo 7.

Capítulo 5

Implementação de um Avaliador Parcial em ASM

O avaliador parcial $\texttt{mix}_{Java}$, apresentado no Capítulo 4, processa regras ASM básicas e foi implementado em Java. Para permitir "auto-aplicação", foi desenvolvida uma segunda versão desse avaliador parcial, implementada na própria linguagem ASM. Essa segunda versão, denominada $\texttt{mix}_{ASM}$, será discutida em detalhes neste capítulo.

O avaliador parcial $\texttt{mix}_{ASM}$ utiliza a abordagem *offline*, assim o processamento é dividido em duas fases: *BTA* e *especialização*. O avaliador parcial recebe como entradas uma especificação ASM e uma definição inicial para o valor das funções estáticas de entrada. Produz como saída uma especificação ASM residual.

Para realizar as experiências com a aplicação da Segunda Projeção de Futamura, envolvendo $\texttt{mix}_{Java}$ e $\texttt{mix}_{ASM}$, bastaria que $\texttt{mix}_{ASM}$ implementasse a fase de especialização. Entretanto, optamos por descrever o processo completo na linguagem ASM, assim temos uma formalização que permite um entendimento claro dos algoritmos e facilidade para demonstrar propriedades sobre seu funcionamento.

Como uma das entradas do avaliador parcial é uma especificação ASM, bem como a saída produzida por ele, vamos discutir na Seção 5.1 como as especificações vão ser internamente representadas. A Seção 5.2 apresenta um pré-processamento executado sobre a especificação. A fase de BTA é discutida na Seção 5.3 e a fase de especialização é o assunto da Seção 5.4. A Seção 5.5 discute aspectos relativos à auto-aplicação e geração de compiladores.

5.1 Representação de Especificações

O avaliador parcial 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 a seguir, onde f denota nomes de funções, t denota termos e R

denota regras:

$$
\begin{array}{rl}
R ::= & f(t_1, \ldots, t_r) := t \\
R ::= & R_1 \ldots R_k \\
R ::= & \texttt{if}\ \ t\ \ \texttt{then}\ \ R_1\ \ \texttt{else}\ \ R_2\ \ \texttt{endif} \\
t ::= & f(t_1, \ldots, t_r)
\end{array}
$$

5.1.1 Especificações ASM de Entrada

Para representar as especificações ASM que servem como entrada para o avaliador parcial, vamos utilizar definições similares às apresentadas em [27]. Uma *especificação de entrada* $\mathcal{S}$, associada a uma álgebra $\mathcal{A}_{in}$, é definida como $\mathcal{S} = (\Upsilon_{in}, \mathcal{D}, \mathit{Init}, \mathit{Prog})$, onde

Υ_{in} é um vocabulário contendo pelo menos todos os nomes de funções que ocorrem em $\mathcal{D}$, *Init* e *Prog*. Algumas das funções são identificadas como *funções de entrada*, ou seja, que definem os valores de entrada para a especificação.

$\mathcal{D}$ é um conjunto de definições de funções. Podem ser definições construtivas, que o avaliador parcial é capaz de avaliar, ou declaração de interfaces, definindo funções externas.

Init é um bloco de regras de atualização que define o estado inicial S_0 de $\mathcal{A}_{in}$.

Prog é a regra de transição.

É interessante utilizar um bloco de regras para representar *Init* porque assim a inicialização pode também ser especializada.

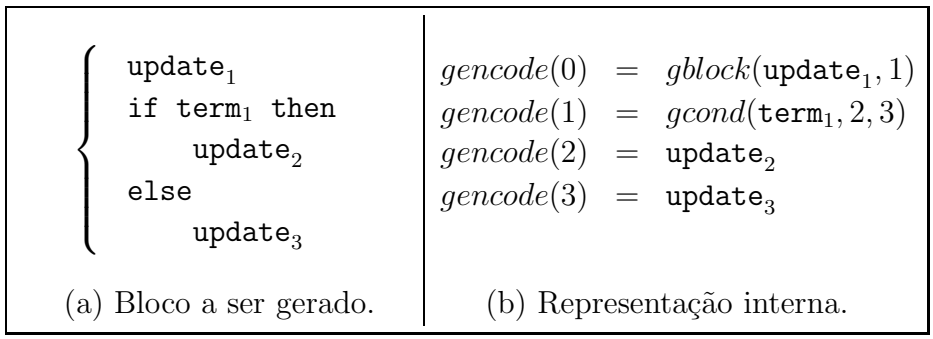
Como em [27], vamos definir um mapeamento "$[\![.]\!]$" que estabelece, para cada objeto sintático, o elemento correspondente em um dos seguintes domínios: *FNAME* (nomes de funções), *DEF* (definições de funções), *TERM* (termos), *RULE* (regras de transição). Vamos assumir que a especificação de entrada está sintaticamente correta e que a codificação é realizada por um pré-processamento da entrada.

Para nomes de funções f de Υ_{in}, temos $[\![f]\!] = \psi(f)$, onde ψ é uma função injetiva de Υ_{in} em *FNAME*. O formato dos elementos dos domínios *TERM* e *RULE* são definidos usando-se construtores, como descrito a seguir:

$$
\begin{array}{rcl}
\mathit{term} & : & (\mathit{FNAME} \times \mathit{TERM}^*) \rightarrow \mathit{TERM} \\
\mathit{update} & : & (\mathit{TERM} \times \mathit{TERM}) \rightarrow \mathit{RULE} \\
\mathit{block} & : & \mathit{RULE}^* \rightarrow \mathit{RULE} \\
\mathit{cond} & : & (\mathit{TERM} \times \mathit{RULE} \times \mathit{RULE}) \rightarrow \mathit{RULE}
\end{array}
$$

5.1.2 Especificações Anotadas

Especificações anotadas são produzidas pela fase de BTA. São muito similares às especificações de entrada, com a exceção de que a cada objeto sintático é associado um valor BTA, que pode ser *BTAPOS* ou *BTANEG*. O valor *BTAPOS* indica que o objeto sintático



(a) Bloco a ser gerado. (b) Representação interna.

Figura 5.1: Exemplo de uso da tabela *gencode*.

pode ser computado durante a especialização, enquanto que *BTANEG* indica um objeto "residualizável".

O universo *TBTA* representa os valores BTA. Termos e regras anotados são elementos dos universos *TTERM* e *TRULE*. Construtores incluindo as anotações são apresentados abaixo. Um construtor especial *lift* é usado para indicar um termo positivo que ocorre em um contexto negativo (ver Seção 5.3).

$$\begin{aligned} tterm &: (TBTA \times FNAME \times TTERM^*) \rightarrow TTERM \\ tupdate &: (TBTA \times TTERM \times TTERM) \rightarrow TRULE \\ tblock &: TRULE^* \rightarrow TRULE \\ tcond &: (TBTA \times TTERM \times TRULE \times TRULE) \rightarrow TRULE \\ lift &: TTERM \rightarrow TTERM \end{aligned}$$

5.1.3 Especificações Residuais

Especificações residuais são elementos do universo *GRULE*. Esses elementos podem ser definidos pelos construtores *term* e *update* apresentados anteriormente, junto com novos construtores *gblock* and *gcond*:

$$\begin{aligned} gblock &: (null \cup (GRULE \times GVAL)) \rightarrow GRULE \\ gcond &: (TERM \times GVAL \times GVAL) \rightarrow GRULE \end{aligned}$$

Um bloco residual pode ser um bloco vazio (*null*) ou uma regra residual, representando a primeira regra do bloco, associada a um valor *GVAL* que indica a próxima regra da seqüência dentro do bloco. Uma regra condicional residual é representada pela guarda (um termo *TERM*) e por dois valores *GVAL* que indicam as cláusulas *then* e *else*.

Uma função *gencode* será utilizada pelo avaliador parcial para estabelecer elos de ligação entre as regras geradas, usando para isso elementos de *GVAL*. Um exemplo é apresentado na Figura 5.1, onde *GVAL* são valores inteiros. A Figura 5.1(a) mostra um bloco com duas regras. Na Figura 5.1(b), a tabela *gencode* é utilizada para armazenar esse bloco.

```
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
```

Figura 5.2: Pré-processamento.

5.2 Pré-Processamento

Para simplificar a codificação do avaliador parcial, a regra de transição das especificações ASM será pré-processada. Esse pré-processamento é o mesmo que foi descrito informalmente na Seção 4.2, no Capítulo 4.

A Figura 5.2 exibe uma definição formal para esse procedimento, usando o paradigma funcional de avaliação estrita. A função `preproc(`r`)` recebe a regra de transição da especificação e retorna uma nova regra, pré-processada. A função auxiliar `merge_blocks`(r_1, r_2) constrói um bloco contendo as regras de r_1 e r_2.

5.3 Análise de Tempo de Definição

A análise de tempo de definição é dividida em dois passos seqüenciais. O primeiro passo consiste em computar uma divisão de todas as funções usadas na especificação, classificando-as como positivas (estáticas) ou negativas (dinâmicas). No segundo passo, uma especificação anotada é gerada.

5.3.1 Computando uma Divisão BTA

Inicialmente, o usuário deve atribuir valores BTA (*BTAPOS* e *BTANEG*) às funções de entrada, indicando em relação a quais entradas a especificação deverá ser especializada. O objetivo desta fase é determinar uma *divisão BTA*, ou seja, classificar as funções não-externas como positivas ou negativas, dependendo da sua relação com as funções de

```
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
```

Figura 5.3: Regras para computar uma divisão BTA.

```
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
```

Figura 5.4: Macro `ProcessBTA(`x`)`, se x é um bloco ou regra condicional.

entrada. Todas as funções externas são classificadas como negativas, e as demais funções são inicialmente classificadas como positivas.

A Figura 5.3 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 *BTAPOS* (estático) e *BTANEG* (dinâmico) no lugar dos reais valores das funções.

O algoritmo de divisão é executado até que um ponto fixo seja alcançado, o que é indicado pela função booleana `change`. A função `init` obtém a regra de inicialização da especificação de entrada, enquanto que `prog` obtém a regra de transição.

A cada passo, uma subregra ou termo é analisado pela "macro" `ProcessBTA`, cuja definição é exibida nas figuras 5.4, 5.5 e 5.6. Todos os termos $f(t_1, \ldots, t_r)$ da especificação são analisados. Se qualquer $t_i, 1 \leq i \leq r$ for negativo, então a função f deve ser classificada como negativa. Além disso, nas atualizações $f(t_1, \ldots, t_r) := t$, se t é negativo, então f deve também ser classificada como negativa.

Uma relação unária `BSET` : $(RULE{+}TERM) \rightarrow BOOL$ é utilizada, de forma similar

```
if x = update (term(f, t*), term(g, u*)) then
  BSET(term(f, t*)) := true
  BSET(term(g, u*)) := true
  if bta_val(g) = BTANEG and bta_val(f) = BTAPOS then
    bta_val(f) := BTANEG
    change := true
  endif
endif
```

Figura 5.5: Macro `ProcessBTA(`x`)`, se x é uma regra de atualização.

```
if x = term (f, ⟨g_1(t_1^*), ..., g_k(t_k^*)⟩) and k > 0 then
  var j ranges over 1..k
    BSET(g_j(t_j^*)) := true
    if bta_val(g_j) = BTANEG and bta_val(f) = BTAPOS then
      bta_val(f) := BTANEG
      change := true
    endif
  endvar
endif
```

Figura 5.6: Macro `ProcessBTA(`x`)`, se x é um termo.

à adotada em [27], para identificar 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 `bta_val` associa, a cada nome de função f de Υ_{in}, um valor *TBTA* (*BTAPOS* ou *BTANEG*). Para cada função f de Υ_{in}, `bta_val(`f`)` é inicializada com *BTAPOS*, exceto para as funções de entrada e funções externas.

A função booleana `change` é utilizada para determinar quando um ponto fixo é alcançado. Seu valor inicial é *false*.

É importante notar que o processo de BTA descrito é *monovariante*. Isto é, a uma função é atribuído um único valor *TBTA*, que vale para todas as suas ocorrências na especificação. Mesmo em especificações simples, essa abordagem pode ser muito restritiva para funções pré-definidas. Por exemplo, todas as ocorrências da função de adição de inteiros seriam negativas (dinâmicas), se existir apenas uma ocorrência de adição envolvendo valor dinâmico. Assim, é interessante adicionar um código que trate de maneira especial as funções pré-definidas, deixando o algoritmo agir da forma descrita nesta seção apenas para as funções criadas pelo usuário.

5.3.2 Gerando uma Especificação Anotada

Depois de computar uma divisão BTA, uma especificação anotada será gerada. As funções utilizadas para construir regras anotadas são exibidas na figuras 5.7 e 5.8. A função `gentr` descreve um mapeamento de regras para regras anotadas, e `gentt` descreve um mapeamento de termos para termos anotados. Uma terceira função `gentt`*, não exibida, é aplicada a listas de termos. A função auxiliar `merge_blocks` tem um comportamento equivalente ao apresentado na Figura 5.2, só que agora atua sobre blocos de regras anotadas.

Se um termo positivo ocorre em um contexto negativo, o resultado da computação deve ser residualizado. Para indicar essa situação, um construtor *lift* é usado na representação anotada. A função `gentt` usa seu segundo argumento, que está relacionado ao contexto onde o termo é avaliado, para determinar se um *lift* é necessário.

Observe que toda atualização anotada é gerada dentro de um bloco, mesmo que o bloco contenha apenas essa atualização. A única razão para isso é produzir uma simplificação no código do algoritmo de especialização.

```
gentr : RULE → TRULE

gentr (block (⟨r_1, ..., r_k⟩)) ≡
  merge_blocks (gentr(r_1), gentr(block (⟨r_2, ..., r_k⟩)))
gentr (cond (c, r_1, r_2)) ≡
  let term(f, t*) = c
      tag= bta_val(f)
  in tcond (tag, gentt(t,tag), gentr(r_1), gentr(r_2))
  endlet
gentr (update (t_1, t_2)) ≡
  let term(f, t*) = t_1
      tag= bta_val(f)
  in tblock(⟨ tupdate (tag, gentt(t_1,tag), gentt(t_2,tag)) ⟩)
  endlet
```

Figura 5.7: Função `gentr`.

5.4 Especialização

O algoritmo de especialização usa a especificação anotada produzida pela fase de BTA e os valores das funções de entrada positivas para gerar uma especificação residual. A técnica utilizada é *especialização polivariante*. O processo consiste em computar o conjunto de todos os *pontos de programa* especializados alcançáveis [55].

Não faz muito sentido falar de pontos de programas, no caso de ASM. Avaliadores parciais para linguagens imperativas ou funcionais consideram separadamente trechos de código do programa a ser especializado, por isso esses trechos são chamados de pontos de programa. Usando os valores estáticos, tem-se pontos de programas especializados. Por outro lado, o avaliador parcial para ASM, em cada iteração, analisa **toda** a regra de transição da especificação a ser especializada. Assim apenas um conjunto de valores associados às funções estáticas (positivas) é importante para identificar "pontos de programa especializados", que nesse caso chamaremos de *estados positivos*. Cada diferente conjunto de valores associados às funções estáticas dá origem a um estado positivo diferente.

Antes de mostrarmos a descrição do algoritmo de especialização, vamos discutir como os estados positivos são representados e computados.

5.4.1 Representação de Estados

Quando o avaliador parcial encontra estruturas dinâmicas, um código residual é gerado. Quando encontra estruturas estáticas, o avaliador parcial age como um meta-interpretador, computando os mesmos resultados que a especificação sendo especializada produziria. Assim, as funções estáticas da especificação de entrada devem ser representadas de alguma forma pelo avaliador parcial, que deve ser capaz também de computar todos os valores estáticos.

```
gentt : TERM × TBTA → TTERM

gentt (term(f, t*), tag) ≡
  let tag_1 = bta_val(f)
      tt = gentt*(t*, tag_1) in
    if (tag= BTANEG) and (tag_1 = BTAPOS) then
      lift(tterm (tag_1, f, tt))
    else
      tterm (tag_1, f, tt)
    endif
  endlet
```

Figura 5.8: Função `gentt`.

Um universo

$$VALUE = BOOL \cup \{undef\} \cup \ldots \cup N$$

é usado para construir interpretações para os nomes de funções estáticas (positivas) de Υ_{in}. Para computar o valor dos termos positivos, as seguintes funções auxiliares são definidas, similares às utilizadas em [27]:

- $g_1, \ldots, g_r$ representam funções predefinidas.
- *apply*: $DEF \times VALUE^* \rightarrow VALUE$ é uma função que produz valores para as funções da especificação de entrada cuja definição é construtiva ou é uma interface para uma função externa. Os parâmetros são uma definição de função e uma seqüência de argumentos.
- `VFuncs` : $LOC \rightarrow VALUE$ define interpretações para nomes de funções estáticas (positivas) da especificação de entrada, onde o universo LOC é (FNAME×TERM*). Por exemplo, se a especificação de entrada contém um nome de função f e em um determinado estado o valor $f(t^*) = v$ é observado, isso é representado pelo avaliador parcial como `VFuncs`$(\langle f, t^* \rangle) = v$. A função `VFuncs` irá também sofrer atualizações que correspondam às atualizações sofridas pelas funções positivas da especificação de entrada.

Interpretações para os termos anotados são fornecidos pela função de avaliação $Val : TTERM \rightarrow VALUE$. Essa função será aplicada somente a termos positivos e é

$$
\begin{array}{l}
\texttt{Reduce}\text{: } \mathit{TTERM} \rightarrow \mathit{TERM} \\
\\
\texttt{Reduce } (\mathit{tterm}\,(\mathit{BTANEG}, f, \langle t_1, \ldots, t_n\rangle)) \equiv \\
\quad \mathit{term}\,(f, \langle \texttt{Reduce}\,(t_1), \ldots, \texttt{Reduce}\,(t_n)\rangle) \\
\texttt{Reduce } (\mathit{tterm}\,(\mathit{BTAPOS}, f, \langle t_1, \ldots, t_n\rangle)) \equiv \\
\quad \mathit{Val}\ (\mathit{tterm}\,(\mathit{BTAPOS}, f, \langle t_1, \ldots, t_n\rangle)) \\
\texttt{Reduce } (\mathit{lift}(t)) \equiv \\
\quad \mathit{term}(\,\mathit{Val}(t))
\end{array}
$$

Figura 5.9: Função `Reduce`.

definida pela equação abaixo:

$$
\begin{array}{l}
\mathit{Val}\ (\mathit{tterm}\,(\mathit{BTAPOS}, f, \langle t_1, \ldots, t_n\rangle)) \equiv \\
\quad \texttt{let } \bar{x} = \langle\, \mathit{Val}\,(t_1), \ldots, \mathit{Val}\,(t_n)\rangle \texttt{ in} \\
\quad\ \texttt{if } f = \text{“}g_1\text{”} \texttt{ then } g_1\,(\bar{x}) \\
\quad\ \ldots \\
\quad\ \texttt{else if } f = \text{“}g_r\text{”} \texttt{ then } g_r\,(\bar{x}) \\
\quad\ \texttt{else if } \mathit{def}(f) \neq \mathit{undef} \texttt{ then } \mathit{apply}(\mathit{def}\,(f), \bar{x}) \\
\quad\ \texttt{else VFuncs } (\langle f, \bar{x}\rangle) \\
\quad\ \texttt{endif} \\
\quad \texttt{endlet}
\end{array}
$$

Quando um termo negativo é processado pelo avaliador parcial, o código residual gerado deve ter todas as informações positivas computadas. A função `Reduce`, exibida na Figura 5.9, produz esses termos residuais, que são chamados de *termos reduzidos*. Para produzir um termo reduzido, termos anotados negativos são simplesmente convertidos a termos sem anotações. Termos positivos são avaliados usando-se a função *Val*. Quando um construtor *lift* é encontrado, denotando um termo positivo inserido em um contexto negativo (ver Seção 5.3), o valor positivo é computado e convertido a um construtor *term*.

O avaliador parcial deve ainda representar de alguma forma todo o conjunto de estados positivos gerados. Esses estados são diferentes instâncias de valores que podem ser associados às funções positivas da especificação de entrada, e serão representados pela função

$$\texttt{TSPEC} : (\mathit{LOC} \times \mathit{INT}) \rightarrow \mathit{VALUE}$$

Essa função é similar a `VFuncs`, com um argumento adicional, que é um valor inteiro. Enquanto `VFuncs` representa as funções positivas da especificação de entrada, `TSPEC` funciona como uma “tabela” de estados positivos, indexada por valores inteiros. Valores inteiros diferentes representam estados positivos diferentes.

```
if (∃x ∈ GSET) then
  choose x ∈ GSET
    GSET(x) := false
    ProcessRule(x)
  endchoose
endif
```

```
if ¬(∃x ∈ GSET) then
  if CurS < TotS then
    CodeNum := 0
    GSET(⟨prog,0,null⟩) := true
    var y ranges over
        Dom(1,VFuncs) ∪ Dom(1,TSpec)
      VFuncs(y) := TSpec(y,CurS+1)
    endvar
    CurS := CurS + 1
  endif
endif
```

Figura 5.10: SPECRULES: Regras para Especialização.

5.4.2 Algoritmo para Especialização

As regras ASM que implementam o algoritmo de especialização são exibidas na Figura 5.10. Uma relação unária GSET identifica as instâncias de subregras que são consideradas em cada passo, de uma maneira similar a BSET na Seção 5.3. A diferença é que os elementos de GSET são uma tupla formada por uma regra, um número inteiro usado como elo de ligação na geração de código residual, e um conjunto de atualizações coletadas.

O valor inicial de GSET é $\{\langle \texttt{init}, 0, \mathit{null}\rangle\}$, onde init é uma função que obtém a regra de inicialização da especificação **anotada**. Observe que a regra de inicialização é processada uma única vez. Todas as iterações seguintes serão executadas sobre a regra de transição prog.

Como explicado na Seção 5.4.1, a função TSPEC representa todo o conjunto de estados positivos gerados, funcionando como uma tabela indexada por valores inteiros. A função inteira de zero argumentos CurS indica qual é o estado positivo corrente, aquele que está sendo processado no momento. A função TotS indica o número total de estados positivos gerados. Os estados com numeração inferior a CurS são os que já foram processados. As funções CurS e TotS são ambas inicializadas com o valor inteiro 0 (zero). A função CodeNum é usada como auxiliar na geração de código residual.

Na Figura 5.10, o comando var utilizado tem o objetivo de atualizar a função VFuncs com os valores associados ao estado positivo corrente, indicado por CurS. O valor inicial de VFuncs associa *undef* a todos os possíveis valores de seu domínio. A construção $\texttt{Dom}(n, f)$ indica o conjunto de todos os valores do n-ésimo domínio de uma função f, para os quais essa função é definida (valor diferente de *undef*).

Quando um estado positivo é processado, regras residuais associadas a ele são geradas e novos estados positivos são produzidos. Esse processo é executado pela macro ProcessRule, cujo funcionamento será discutido nas próximas seções. Os estados positivos que ainda não tiverem sido processados são inseridos na "tabela" TSPEC. O algoritmo é executado enquanto existirem estados positivos não processados. Após o final da execução, uma regra residual R_k estará associada a cada diferente estado positivo k. A especificação residual conterá uma função adicional curstate e a regra de transição residual completa

```
let ⟨tblock(r_1, ..., r_k), cn, updates⟩ = x in
  if k > 0 then
    ProcessUpdate (r_1, cn, tblock(r_2, ..., r_k), updates)
  else    // ...é um bloco vazio
    var u ranges over updates
      let ⟨tterm (BTAPOS, f, ⟨t_1, ..., t_j⟩), v⟩ = u in
        VFuncs (⟨f, ⟨Val(t_1), ..., Val(t_j)⟩⟩) := Val(v)
      endlet
    endvar
    CN := cn
    BuildNewState := 1
  endif
endlet
```

Figura 5.11: Regra `ProcessRule(`x`)`, para blocos de regras anotadas.

será um bloco com regras da forma "`if curstate` $= k$ `then` R_k". Esse formato da regra de transição residual completa será melhor explicado na Seção 5.4.7.

5.4.3 Processando Blocos

Como explicado na Seção 5.4.2, os elementos de `GSET` são uma tupla composta por uma regra anotada, um número inteiro usado na geração de código, e uma seqüência de atualizações coletadas. A seqüência de atualizações coletadas é formada somente por objetos sintáticos, onde cada atualização é representada por um par de elementos *TTERM*.

A Figura 5.11 mostra um algoritmo para processar elementos inseridos em `GSET`, para o caso em que esses elementos estão associados a um *bloco* de regras anotadas. Regras condicionais e atualizações serão discutidas nas seções seguintes.

Se a regra anotada é um bloco, então será sempre um bloco que contém apenas regras de atualização anotadas, devido ao processo de construção do programa anotado, descrito na Seção 5.3.2. Em cada iteração, a primeira atualização do bloco é processada, inserindo-se novos elementos em `GSET` (ver descrição de `ProcessUpdate` na Seção 5.4.4).

Se o bloco é vazio, um novo estado positivo é produzido, disparando-se em paralelo as atualizações coletadas. Isso é representado, na Figura 5.11, por um comando `var` que constrói um estado em `VFuncs`. Observe que a função *Val* deve ser executada sobre cada um dos termos, pois as atualizações coletadas são constituídas apenas de elementos sintáticos. Após a construção de um estado positivo em `VFuncs`, deve-se determinar se esse estado já está armazenado em `TSPEC` e gerar um código residual apropriado. Essas ações serão realizadas em mais de um passo do algoritmo, alterando inclusive o código do corpo do algoritmo de especialização descrito na Seção 5.4.2. Explicaremos esse processo mais tarde, na Seção 5.4.6. Por enquanto, basta saber que o valor inteiro usado na geração de código residual é armazenado na função `CN`, e que a função `BuildNewState` controla a seqüência de ações a ser executada.

```
let tupdate (tag, t1, t2) = r in
  if tag = BTAPOS then
    let newupdates = cons (⟨t1, t2⟩, updates) in
      GSET (⟨rest, cn, newupdates⟩) := true
    endlet
  else
    GSET (⟨rest, CodeNum+1, updates⟩) := true
    CodeNum := CodeNum + 1
    gencode(⟨CurS,cn⟩) := gblock (
      update (Reduce (t1), Reduce (t2)), ⟨CurS,CodeNum+1⟩ )
  endif
endlet
```

Figura 5.12: `ProcessUpdate(`r`,cn,`$rest$`,updates)`.

5.4.4 Processando Regras de Atualização

A Figura 5.12 exibe o detalhamento da macro `ProcessUpdate`. Essa macro processa regras de atualização anotadas com um formato $tupdate(tag, t_1, t_2)$.

Se a regra é positiva, a nova subregra inserida em `GSET` é formada pelo resto do bloco corrente, com $\langle t_1, t_2 \rangle$ adicionado ao conjunto de atualizações coletadas. O número de código é o mesmo do bloco corrente, porque nenhum código residual é gerado. Observe que t_1 e t_2 são objetos sintáticos.

Se a regra é negativa, a nova subregra inserida em `GSET` é formada pelo resto do bloco corrente, com o mesmo conjunto de atualizações coletadas. Um novo número de código é atribuído a essa subregra, porque um bloco residual é gerado usando o número de código corrente. A função `gencode`, como explicado na Seção 5.1, associa uma regra residual a cada bloco processado, usando valores do universo *GVAL* para identificar unicamente cada bloco gerado. Nesse caso, elementos do universo *GVAL* são representados por um par: o valor de `CurS`, que indica o estado positivo corrente, e um número de código `cn`. Uma versão reduzida (ou seja, com informação positiva computada) da regra de atualização é a primeira regra do bloco residual gerado. As regras seguintes desse bloco residual ainda serão geradas, quando a nova subregra inserida em `GSET` for processada.

5.4.5 Processando Regras Condicionais

Regras condicionais anotadas $tcond(tag, cond, rthen, relse)$ são processadas pelo avaliador parcial seguindo o algoritmo definido pelas regras ASM exibidas na Figura 5.13.

Se a regra é positiva, então a condição é avaliada usando-se a função *Val*. Dependendo do resultado (*true* ou *false*), a nova subregra a ser processada no próximo passo será *rthen* or *relse*.

Se a regra é negativa, duas novas subregras são inseridas em *GSET*. Observe que novos números de código são associados a essas subregras. O código residual gerado é uma regra

```
let ⟨tcond (tag, cond, rthen, relse), cn, updates⟩ = x in
  if tag = BTAPOS then
    if Val(cond) then
      GSET (⟨rthen, cn, updates⟩) := true
    else
      GSET (⟨relse, cn, updates⟩) := true
    endif
  else
    GSET (⟨rthen, CodeNum+1, updates⟩) := true
    GSET (⟨relse, CodeNum+2, updates⟩) := true
    CodeNum := CodeNum + 2
    gencode(⟨CurS,cn⟩) := gcond (Reduce(cond),
      ⟨CurS,CodeNum+1⟩, ⟨CurS,CodeNum+2⟩)
  endif
endlet
```

Figura 5.13: Regra `ProcessRule(`x`)`, para regras condicionais anotadas.

condicional que referencia os números de código dessas subregras. Toda informação positiva da condição residual é computada pela função *Reduce*.

5.4.6 Geração de Novos Estados Positivos

Na Seção 5.4.3, exibimos um algoritmo para processar elementos de `GSET` para o caso em esses elementos estão associados a blocos de regras anotadas. Vimos que, quando o bloco se torna vazio, um novo estado positivo é produzido, disparando-se em paralelo as atualizações coletadas. Após a construção de um estado positivo em `VFuncs`, deve-se determinar se esse estado já está armazenado em `TSPEC` e gerar um código residual apropriado. Dissemos que essas ações são realizadas em mais de um passo do algoritmo, alterando inclusive o código do corpo do algoritmo de especialização descrito na Seção 5.4.2.

O novo corpo do algoritmo de especialização é exibido na Figura 5.14. O código exibido na Figura 5.10, que chamamos de `SPECRULES`, é executado quando o valor de `BuildNewState` é 0 (zero). Esse é exatamente o valor com que a função `BuildNewState` é inicializada. Esse valor é alterado para 1 na Figura 5.11, indicando o início das ações que determinam se o estado positivo gerado `VFuncs` já está armazenado em `TSPEC`.

Cada diferente valor inteiro associado ao segundo argumento de `TSpec` indica um estado positivo diferente. Uma função auxiliar `verif` é utilizada para indicar qual estado positivo armazenado em `TSpec` é equivalente ao estado `VFuncs`. Se o algoritmo funcionar corretamente, apenas um ou nenhum dos estados de `TSpec` será equivalente a `VFuncs`. Inicialmente, `verif` é marcada com *true* para todos os estados de `TSpec`. No passo seguinte (`BuildNewState` = 2), os estados não equivalentes a `VFuncs` são marcados com *false*. Isso é realizado pelos dois comandos `var` aninhados.

A função `NewState` irá armazenar o número inteiro associado ao estado de `TSpec` equi-

```
if BuildNewState = 0 then
  SPECRULES
elseif BuildNewState = 1 then
  var z ranges over Dom(2,TSpec)
    verif(z) := true
  endvar
  BuildNewState := 2
elseif BuildNewState = 2 then
  var z ranges over Dom(2,TSpec)
    var y ranges over Dom(1,VFuncs) ∪ Dom(1,TSpec)
      if VFuncs(y) ≠ TSpec(y,z) then
        verif(z) := false
      endif
    endvar
  endvar
  NewState := TotS + 1
  BuildNewState := 3
elseif BuildNewState = 3 then
  var z ranges over Dom(2,TSpec)
    if verif(z) then
      NewState := z
    endif
  endvar
  BuildNewState := 4
elseif BuildNewState = 4 then
  gencode(⟨CurS,CN⟩) := GenNextStep (NewState)
  if NewState > TotS then
    var y ranges over Dom(1,VFuncs) ∪ Dom(1,TSpec)
      TSpec(y,TotS+1) := VFuncs(y)
    endvar
    TotS := TotS + 1
  endif
  var y ranges over Dom(1,VFuncs) ∪ Dom(1,TSpec)
    VFuncs(y) := TSpec(y,CurS)
  endvar
  BuildNewState := 0
endif
```

Figura 5.14: Algoritmo de especialização revisitado, incluindo geração de novos estados positivos.

valente a `VFuncs`. Quando `BuildNewState` = 2, presume-se que nenhum estado de `TSpec` é equivalente a `VFuncs`, assim `NewState` é atualizada com `TotS`+1. Quando `BuildNewState` = 3, o valor de `NewState` é alterado, caso `verif` ainda seja *true* para algum estado de `TSpec`.

Finalmente, quando `BuildNewState` = 4, um código residual apropriado é gerado, usando o número de código `CN` determinado na Figura 5.11. As regras `SPECRULES` serão novamente executadas, uma vez que `BuildNewState` se torna novamente 0. A função `GenNextStep(v)` gera o código "`curstate := v`", que define o fluxo de controle na regra de transição residual (ver Seção 5.4.7).

5.4.7 A Regra de Transição Residual

Após o último passo do algoritmo de especialização, os elos de ligação estabelecidos pela função `gencode` podem ser utilizados para se construir a regra de transição residual. Cada estado positivo identificado por $k \in$ `Dom`(2,`TSPec`) possui uma regra residual R_k associada, definida por `gencode`$(\langle k, 0\rangle)$.

Uma função adicional `curstate` : INT define o fluxo de controle na especificação residual. 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 $k \in$ `Dom`(2,`TSPec`). O fluxo de controle é determinado por atualizações sobre `curstate`, as quais são geradas quando um bloco vazio é processado pelo algoritmo de especialização (ver Figura 5.14).

5.5 Auto-Aplicação e Geração de Compiladores

A abordagem *offline* simplifica o processo de auto-aplicação de um avaliador parcial porque divide a avaliação parcial em duas fases separadas.

O avaliador parcial $\texttt{mix}_{ASM}$ é composto de dois programas separados: *BTA* (análise de tempo de definição) e *Spec* (especialização). Usando esses programas, a Segunda Projeção de Futamura pode ser reescrita da seguinte forma:

$$\begin{aligned} Spec^{ann} &= [\![BTA]\!](Spec, div_{Spec}) \\ int^{ann} &= [\![BTA]\!](int, div_{int}) \\ compiler &= [\![Spec]\!](Spec^{ann}, int^{ann}) \end{aligned}$$

onde p^{ann} denota uma versão anotada do programa p and div_p denota a divisão das funções de entrada de p em estáticas (positivas) e dinâmicas (negativas).

O processo de geração de compiladores usando o avaliador parcial offline para ASM é conduzido em três etapas. Primeiro, um interpretador para uma linguagem L é escrito em ASM. Segundo, uma versão anotada desse interpretador é gerada usando os algoritmos de BTA descritos na Seção 5.3. Finalmente, uma versão previamente anotada do algoritmo de especialização *Spec* (descrito na Seção 5.4) é especializada com respeito ao interpretador anotado. Um compilador de L para ASM é gerado. Note que a fase de *BTA* não é incluída na auto-aplicação do especializador.

O algoritmo de especialização descrito na Seção 5.4 processa somente regras ASM básicas. O texto da descrição do algoritmo utiliza construções simples de casamento de padrões e expressões `let` para facilitar a codificação e facilitar o entendimento. Essas estruturas podem ser facilmente traduzidas em funções apropriadas que extraiam os componentes desejados. O construtor `choose` também é utilizado, mas sua semântica no algoritmo é essencialmente determinística. Pode ser traduzido em operações que extraem elementos de uma lista. Assim, a descrição inteira poderia ser traduzida em uma especificação que usa somente regras básicas, mas isso não é possível devido a algumas ocorrências de comandos var que não podem ser substiuídos por regras básicas de maneira satisfatória. Isso acontece especialmente no disparo das atualizações coletadas em paralelo (Figura 5.11), impedindo que uma real auto-aplicação seja possível.

Nem todos os programas são escritos de modo adequado à avaliação parcial. As estruturas dependendo de valores estáticos (positivos) e dinâmicos (negativos) devem ser cuidadosamente separadas. As estruturas positivas vão ser computadas durante o tempo de especialização e não vão aparecer no código residual. Para mostrar que o especializador apresentado na Seção 5.4 é adequado à avaliação parcial, é necessário analisar sua versão anotada, produzida pela submissão de seu código ao algoritmo de BTA.

O especializador possui dois dados de entrada: uma especificação anotada e os valores das funções de entrada positivas. Se o especializador vai ser especializado com respeito à primeira entrada, todas as estruturas que dependam apenas da especificação anotada deverão ser classificadas como positivas pela BTA, e as demais serão negativas.

Para acessar a especificação anotada, funções como `init` (regra de inicialização), `prog` (regra de transição) e `def` (definições) são utilizadas. Como dependem apenas da primeira entrada, essas funções serão marcadas como positivas pela BTA. A segunda entrada compreende os valores das funções de entrada positivas. Vamos convencionar que esses valores são acessados por meio de funções externas, as quais são acessadas por meio da função auxiliar *apply* (ver Seção 5.4.1). Assim a função *apply* é marcada como negativa pela BTA.

A função *Val* depende da função *apply*, assim qualquer ocorrência de *Val* é negativa. Outras funções negativas: `CurS`, `TotS` e `gencode`.

Por outro lado, os componentes dos elementos inseridos em `GSET` são todos positivos: as subregras e as atualizações coletadas são extraídos da especificação anotada, e o número de código depende apenas do seu próprio valor anterior. Assim `GSET` é classificado como positivo, como também é positivo o comando var que dispara as atualizações coletadas em paralelo. Agora fica clara a razão de usar apenas objetos sintáticos para representar as atualizações coletadas. Outra possibilidade, como a adotada em [27], seria utilizar os valores já computados, mas isso iria fazer com que a BTA classificasse `GSET` como negativo.

Um compilador residual gerado pela Segunda Projeção de Futamura, aplicando $\texttt{mix}_{Java}$ a $\texttt{mix}_{ASM}$, não possui nenhuma ocorrência das funções positivas enumeradas acima. As estruturas positivas e negativas são satisfatoriamente bem separadas no código do especializador, assim ele pode ser considerado adequado à auto-aplicação.

5.6 Conclusão

A especificação ASM do avaliador parcial apresentada é muito simples e é expressa em poucas linhas de código. Decidimos usar a abordagem offline porque esta simplifica a auto-aplicação [55].

O avaliador parcial para ASM difere de avaliadores parciais para linguagens imperativas e funcionais em vários aspectos. Por exemplo, deve lidar com atualizações paralelas de funções e tem um conceito diferente de *pontos de programa* especializados.

Em linguagens imperativas, um ponto de programa especializado é definido por um par $\langle l, vv \rangle$, onde l é um rótulo que define um ponto do código e vv representa os valores das variáveis estáticas. Em linguagens funcionais, pontos de programa especializados são definidos por um nome de função e valores dos argumentos estáticos. Em especificações ASM, por outro lado, a regra de transição inteira é processada para cada diferente conjunto de funções positivas (estáticas). Assim um ponto de programa especializado é representado somente pelos valores das funções positivas, não necessitando de nada que identifique um ponto do código. Processar a regra de transição inteira a cada passo traz dificuldades adicionais, pois vários novos pontos de programa podem ser produzidos a cada iteração.

Encontramos algumas dificuldades para representar o código gerado pelo especializador. Especificações residuais geradas poderiam ser representadas internamente pelas mesmas estruturas usadas para as especificações de entrada, já que ambas representam código ASM sem anotações. Entretanto, o processo de geração *top-down* usado pelo especializador, sem recursividade, torna o uso dessas estruturas inadequado, especialmente para blocos e regras condicionais. A técnica descrita na Seção 5.1.3 é apropriada para essa geração top-down sem uso de recursividade.

Muitas otimizações podem ser implementadas no avaliador parcial. Uma das mais importantes é a *compressão de transições* [55]. Essa técnica de otimização permite que regras compatíveis possam ser combinadas e regras residuais desnecessárias possam ser eliminadas, produzindo uma especificação mais eficiente.

A auto-aplicação de um avaliador parcial permite a geração de compiladores, mas impõe restrições adicionais à especificação. O avaliador parcial deve processar seu próprio código. Estruturas estáticas (positivas) e dinâmicas (negativas) devem ser cuidadosamente separadas.

Mostramos que as estruturas estáticas e dinâmicas estão razoavelmente bem separadas no código do avaliador parcial. Entretanto, foi necessário utilizar a regra `var` para descrever algumas partes do algoritmo, especialmente o disparo paralelo das atualizações coletadas, de modo que a avaliação parcial do próprio avaliador produzisse resultados satisfatórios. Entretanto, não fomos capazes de descrever o processamento de regras `var` em $\texttt{mix}_{ASM}$ sem afetar de maneira negativa a separação entre as estruturas estáticas e dinâmicas. Para obter resultados eficientes na "auto-aplicação", foi necessário então utilizar um avaliador parcial mais poderoso para especializar o código de $\texttt{mix}_{ASM}$.

Seguindo essa idéa, alguns experimentos envolvendo geração de compiladores e avaliação parcial são descritos em [31]. Usando um avaliador parcial $\texttt{mix}_{Java}$ mais poderoso, implementado em Java, o código de $\texttt{mix}_{ASM}$ foi especializado com respeito a um interpretador para máquina de Turing. O compilador residual produzido processa programas

escritos na linguagem da máquina de Turing e gera código em ASM. Mesmo usando os dois avaliadores parciais, o código produzido ainda não foi muito eficiente, principalmente pela falta de otimizações em $\texttt{mix}_{ASM}$.

A aplicação da Terceira Projeção de Futamura, para produzir um gerador de compiladores para ASM, tornou-se impraticável. Uma possibilidade seria utilizar a seguinte equação:

$$cogen = [\![\texttt{mix}_{Java}]\!](\texttt{mix}_{ASM}, \texttt{mix}_{ASM})$$

mas isso não é possível porque $\texttt{mix}_{ASM}$ não processa comandos `var`, assim não pode ser fornecido como entrada para si próprio.

Devido a todos esses problemas enfrentados, decidimos adotar uma outra abordagem para geração de compiladores. Essa abordagem, conhecida como abordagem de *extensões de geração*, é discutida no Capítulo 7.

Capítulo 6

Propriedades Importantes do Avaliador Parcial

Neste capítulo, vamos provar propriedades interessantes sobre os algoritmos apresentados no Capítulo 5. Os algoritmos foram codificados usando o próprio modelo ASM, mas o paradigma funcional com avaliação estrita também foi utilizado para especificar algumas funções. Vamos mostrar que o modelo ASM torna as demonstrações simples e diretas.

Algumas suposições serão estabelecidas de modo informal, para facilitar as demonstrações. As propriedades que abordaremos estão relacionadas principalmente à terminação dos algoritmos e à semântica das especificações ASM geradas.

Antes de iniciarmos as demonstrações, vamos recordar quais são as entradas para o avaliador parcial apresentado no Capítulo 5:

1. uma especificação ASM $\mathcal{S}$;
2. indicação de quais funções de entrada de $\mathcal{S}$ são estáticas e quais são dinâmicas, juntamente com valores fornecidos para as funções estáticas.

Na Seção 5.1, a tupla $(\Upsilon_{in}, \mathcal{D}, \mathit{Init}, \mathit{Prog})$ é usada para definir uma especificação $\mathcal{S}$. O componente *Init* é um conjunto de regras de atualização que define o estado inicial, e *Prog* é a regra de transição. As regras são codificadas usando os construtores *term*, *update*, *cond* e *block*.

6.1 Pré-Processamento

Primeiro, vamos considerar o pré-processamento de regra de transição, descrito formalmente pela função `preproc`, na Figura 5.2. O algoritmo descrito recebe como entrada uma regra ASM e retorna uma nova regra ASM, pré-processada.

Definition 15 *A função Nconds(r) indica o número de regras condicionais encontradas*

em uma regra ASM r. É definida da seguinte forma:

$$\begin{aligned} Nconds\ (cond(c, r_1, r_2)) &\equiv Nconds(r_1) + Nconds(r_2) + 1 \\ Nconds\ (block(\langle r_1, \ldots, r_k \rangle)) &\equiv Nconds(r_1) + \ldots + Nconds(r_k) \\ Nconds\ (update(loc, val)) &\equiv 0 \end{aligned}$$

A função `merge_blocks`(r_1, r_2) constrói um bloco contendo as regras encontradas em r_1 e r_2; assim, a seguinte relação é válida:

$$Nconds\ (\texttt{merge\_blocks}(r_1, r_2)) \equiv Nconds(r_1) + Nconds(r_2)$$

6.1.1 Terminação do Pré-Processamento

Vamos utilizar a função *Nconds* para demonstrar que o pré-processamento descrito pela função `preproc` sempre termina. Para isso, basta mostrarmos que, em uma chamada de função `preproc`(r), toda subchamada recursiva `preproc`(r') satisfaz à seguinte propriedade:

$$Nconds(r') < Nconds(r)$$

Como $Nconds(Prog)$ é um número finito, e `preproc`(r) termina sempre que $Nconds(r) = 0$, então `preproc`$(Prog)$ irá sempre terminar.

Theorem 1 *Uma chamada de função* `preproc`$(Prog)$ *sempre termina, onde Prog é a regra de transição de uma especificação ASM $\mathcal{S}$.*

Prova: A prova é conduzida sobre a definição de `preproc`(r), mostrando que toda chamada recursiva da função é realizada sobre uma regra com valor de *Nconds* inferior ao argumento de entrada r:

- `preproc` $(update\ (loc,val))$ é um caso trivial.
- Em `preproc` $(cond\ (c, r_1, r_2))$, temos

  $$\begin{aligned} Nconds\ (r_1) &< Nconds\ (cond\ (c, r_1, r_2)) \\ Nconds\ (r_2) &< Nconds\ (cond\ (c, r_1, r_2)) \end{aligned}$$

  por definição de *Nconds*.
- Em `preproc` $(block\ (r_1, \ldots, r_k))$, se não existe regra condicional no bloco, este mesmo bloco é retornado (caso trivial). Se existe uma regra condicional $r_j = cond\ (c, s_1, s_2)$ no bloco, temos

  $$\begin{aligned} Nconds(block(r_1, \ldots, r_k)) = 1 &+ Nconds(s_1) + Nconds(s_2) + \\ &Nconds(block(\langle r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_k \rangle)) \end{aligned}$$

  Como

  $$\begin{aligned} s_3 &= \texttt{merge\_blocks}(s_1,\ block\ (\langle r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_k \rangle)) \\ s_4 &= \texttt{merge\_blocks}(s_2,\ block\ (\langle r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_k \rangle)) \end{aligned}$$

então

$$\begin{aligned} Nconds\ (s_3) &= Nconds(s_1) + Nconds(block(\langle r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_k\rangle)) \\ Nconds\ (s_4) &= Nconds(s_2) + Nconds(block(\langle r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_k\rangle)) \end{aligned}$$

logo

$$\begin{aligned} Nconds\ (s_3) &< Nconds\ (block\ (\langle r_1, \ldots, r_k\rangle)) \\ Nconds\ (s_4) &< Nconds\ (block\ (\langle r_1, \ldots, r_k\rangle)) \end{aligned}$$

usando a definição de *Nconds* e o comportamento dessa função sobre `merge_blocks`.

■

Se *Prog* é a regra de transição de uma especificação ASM $\mathcal{S}$, mostramos que a função `preproc`(*Prog*) sempre termina.

6.1.2 Manutenção da Semântica Original

Vamos demonstrar agora que as transformações produzidas pela função `preproc` não alteram a semântica de *Prog*.

Theorem 2 *Se r é uma regra de transição ASM, então* `preproc`(r) *tem a mesma semântica que r.*

Vamos utilizar a simbologia $r_1 \overset{\text{ASM}}{\approx} r_2$ para indicar que as regras r_1 e r_2 possuem a mesma semântica ASM. Assim, o Teorema 2 pode ser expresso da forma a seguir:

$$\texttt{preproc}(r) \overset{\text{ASM}}{\approx} r$$

Prova: A prova utiliza indução matemática sobre o valor $Nconds(r)$, ou seja, o número de regras condicionais presentes na regra ASM r. Para cada chamada `preproc`(r), vamos supor que $Nconds(r) = k$. A hipótese de indução será a seguinte:

$$\texttt{preproc}(r') \overset{\text{ASM}}{\approx} r'$$

onde r' é qualquer regra ASM com $Nconds(r') < k$. Vamos analisar cada caso separadamente, incluindo os casos em que $Nconds(r) = 0$:

- `preproc`(*update*(*loc*,*val*)) retorna a própria regra de atualização, logo a semântica é obviamente a mesma.
- Em `preproc`$(cond(c, r_1, r_2))$, é facil ver que

$$\begin{aligned} Nconds(r_1) &< Nconds(cond(c, r_1, r_2)) \\ Nconds(r_2) &< Nconds(cond(c, r_1, r_2)) \end{aligned}$$

Assim podemos utilizar a hipótese de indução e afirmar que

$$\texttt{preproc}(r_1) \overset{\text{ASM}}{\approx} r_1$$
$$\texttt{preproc}(r_2) \overset{\text{ASM}}{\approx} r_2$$

Usando as definições de semântica de ASM apresentadas no Capítulo 2, pode-se concluir que

$$cond(c, \texttt{preproc}(r_1), \texttt{preproc}(r_2)) \overset{\text{ASM}}{\approx} cond(c, r_1, r_2)$$

- Em $\texttt{preproc}(block(r_1, \ldots, r_k))$, se não existe uma regra condicional no bloco, o próprio bloco é retornado, assim a semântica é obviamente a mesma.

  Se existe uma regra condicional $r_j = cond(c, s_1, s_2)$ no bloco, vimos no Teorema 1 que

  $$\begin{aligned} Nconds(s_3) &< Nconds(block(\langle r_1, \ldots, cond(c, s_1, s_2), \ldots, r_k\rangle)) \\ Nconds(s_4) &< Nconds(block(\langle r_1, \ldots, cond(c, s_1, s_2), \ldots, r_k\rangle)) \end{aligned}$$

  Usando a hipótese de indução, podemos afirmar que:

  - $\texttt{preproc}(s_3)$ tem a mesma semântica que a execução simultânea das regras s_1 e $r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_k$.
  - $\texttt{preproc}(s_4)$ tem a mesma semântica que a execução simultânea das regras s_2 e $r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_k$.

  Então podemos concluir que

  $$cond(c, \texttt{preproc}(s_3), \texttt{preproc}(s_4)) \overset{\text{ASM}}{\approx} block(\langle r_1, \ldots, r_k\rangle)$$

  pois em $cond(c,\texttt{preproc}(s_3),\texttt{preproc}(s_4))$:

  - a regra s_1 só será executada se a condição c for satisfeita;
  - a regra s_2 só será executada se a condição c não for satisfeita;
  - as regras $r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_k$ serão sempre executadas, independente do resultado da avaliação da condição c.

  Ou seja, $cond(c,\texttt{preproc}(s_3),\texttt{preproc}(s_4))$ possui exatamente a mesma semântica de $block(\langle r_1, \ldots, r_k\rangle)$.

■

Nesta seção, analisamos o algoritmo de pré-processamento apresentado na Seção 5.2. Esse algoritmo modifica uma especificação ASM de entrada, com o objetivo de simplificar a codificação do avaliador parcial. Mostramos que o algoritmo sempre termina, e que o mesmo não altera a semântica da especificação original.

$$
\begin{array}{l}
\texttt{progToN} : (\mathit{RULE} + \mathit{TERM}) \rightarrow N \\
\\
\texttt{progToN}\ (\mathit{update}\ (t_1, t_2)) \equiv \\
\quad 1 + \texttt{progToN}(t_1) + \texttt{progToN}(t_2) \\
\texttt{progToN}\ (\mathit{cond}\ (c, r_1, r_2)) \equiv \\
\quad 1 + \texttt{progToN}(c) + \texttt{progToN}(r_1) + \texttt{progToN}(r_2) \\
\texttt{progToN}\ (\mathit{block}\ (\langle r_1, \ldots, r_k \rangle)) \equiv \\
\quad 1 + \texttt{progToN}(r_1) + \ldots + \texttt{progToN}(r_k) \\
\texttt{progToN}\ (\mathit{term}\ (f, \langle t_1, \ldots, t_k \rangle)) \equiv \\
\quad 1 + \texttt{progToN}(t_1) + \ldots + \texttt{progToN}(t_k)
\end{array}
$$

Figura 6.1: Função `progToN`.

6.2 Análise de Tempo de Definição

O primeiro passo da fase de BTA consiste em computar uma divisão de todas as funções usadas na especificação, classificando-as como positivas (estáticas) ou negativas (dinâmicas).

O usuário estabelece uma classificação para as funções de entrada da especificação, e o algoritmo de BTA exibido nas Figuras 5.3, 5.4, 5.5 e 5.6 se encarrega de classificar as demais funções. A classificação é representada pela função

$$\texttt{bta\_val} : \mathit{FNAME} \rightarrow \mathit{TBTA}$$

onde *TBTA* é um universo contendo os valores *BTAPOS* e *BTANEG*. Com exceção dos nomes das funções de entrada, cuja classificação é definida pelo usuário, e das funções externas, `bta_val` inicialmente associa o valor *BTAPOS* a todos os nomes de funções da especificação.

6.2.1 Terminação da BTA

Primeiramente, vamos procurar demonstrar que o algoritmo de BTA sempre termina. A função `progToN` irá nos ajudar nessa demonstração. Essa função é definida na Figura 6.1, onde N é o conjunto dos números naturais. A função `progToN` associa números naturais a elementos sintáticos usados para representar uma regra ASM. Uma propriedade interessante dessa função é que o número associado a um elemento sintático é maior que a soma dos números associados aos seus componentes.

O corpo do algoritmo de BTA é exibido na Figura 5.3. O primeiro conjunto de regras, na parte esquerda da Figura 5.3, será executado se o conjunto `BSET` for não vazio. O segundo conjunto de regras, na parte direita da mesma figura, só será executado quando `BSET` for vazio e o valor da função `change` for *true*. Na Seção 5.3, foi estabelecido que o valor inicial de `BSET` é $\{\texttt{init}, \texttt{prog}\}$, onde `init` representa a regra de inicialização e `prog`

representa a regra de transição da especificação de entrada. Assim, no primeiro passo da execução, o primeiro conjunto de regras será disparado.

Antes de mostrar que o algoritmo de BTA sempre pára, vamos mostrar que, em um número finito de passos, o conjunto BSET se tornará vazio. Isso deverá acontecer não apenas a partir do primeiro passo da execução, mas a partir de qualquer estado onde BSET seja não vazio.

Seja *SumBSET* a seguinte função, onde $t \in TERM$, $f \in FNAME$:

$$\begin{aligned} SumBSET &: 2^{(RULE+TERM)} \rightarrow N \\ SumBSET\ (\{x_1, \ldots, x_k\}) &\equiv \sum_{j=1}^{k} \texttt{progToN}\ (x_j) \end{aligned}$$

O conjunto BSET, como definido na Seção 5.3, contém elementos que representam regras e termos ASM. A função *SumBSET* representa o somatório de progToN aplicado a cada elemento de um conjunto como BSET. Obviamente, $SumBSET(\emptyset) = 0$.

Suponha que o conjunto BSET seja não vazio. Então o primeiro conjunto de regras da Figura 5.3 é executado. Um elemento x é eliminado de BSET e processado pela macro ProcessBTA, exibida nas Figuras 5.4, 5.5 e 5.6. Suponha que BSET′ seja o novo valor de BSET, no passo seguinte. Vamos analisar cada caso separadamente:

- Se $x = block(r_1, \ldots, r_k)$, então k novas regras ASM são inseridas em BSET. Nesse caso, $SumBSET(\texttt{BSET}') = SumBSET(\texttt{BSET}) - 1$.
- Se $x = cond(c, r_1, r_2)$, então c, r_1 e r_2 são inseridos em BSET. Novamente, $SumBSET(\texttt{BSET}') = SumBSET(\texttt{BSET}) - 1$.
- Se $x = update(t_1, t_2)$, então dois termos são inseridos em BSET. Mais uma vez, $SumBSET(\texttt{BSET}') = SumBSET(\texttt{BSET}) - 1$.
- Finalmente, se $x = term\ (f, \langle t_1, \ldots, t_k \rangle)$, k novos termos são inseridos em *BSET*. Então $SumBSET(\texttt{BSET}') = SumBSET(\texttt{BSET}) - 1$, como nos demais casos.

O valor inicial de BSET é $\{\texttt{init}, \texttt{prog}\}$, que é o mesmo valor que essa função assume quando o segundo conjunto de regras da Figura 5.3 é executado. Nesse caso, $SumBSET(\texttt{BSET}) = \texttt{progToN}(\texttt{init}) + \texttt{progToN}(\texttt{prog})$. Pela análise descrita acima, podemos observar o seguinte: a cada passo da execução, se o conjunto BSET é não vazio, o valor de *SumBSET* aplicado ao mesmo é essencialmente decrescente. Quando o valor de *SumBSET* é zero, isso significa que o conjunto se tornou vazio.

Theorem 3 *Suponha que S_k seja um estado de uma execução do algoritmo de BTA da Seção 5.3. Se a guarda da regra exibida no lado esquerdo da Figura 5.3 é satisfeita no estado S_k, então essa guarda será falsa em um número finito de passos após S_k.*

Prova: Vamos supor que o algoritmo de BTA seja executado com um estado inicial S_0 onde BSET é $\{\texttt{init}, \texttt{prog}\}$, e que em um determinado passo a guarda da regra exibida no lado esquerdo da Figura 5.3 é satisfeita. O valor máximo que $SumBSET(\texttt{BSET})$ pode

atingir é `progToN(init)`+ `progToN(prog)`, um número finito. De acordo com a análise conduzida acima, o valor de *SumBSET*(`BSET`) é necessariamente decrescente a cada passo da execução, assim eventualmente será zero, significando que `BSET` tornou-se um conjunto vazio. Logo, a guarda eventualmente não será satisfeita. ■

Observe uma importante diferença entre as demonstrações dos Teoremas 1 e 3. O primeiro trata de uma função definida recursivamente, assim a demonstração de terminação se baseia nas chamadas recursivas. O segundo está associado a uma especificação ASM, assim foi necessário estabelecer uma propriedade sobre o estado geral e mostrar que um valor decrescente é sempre gerado, para todos os casos possíveis da regra ASM.

O Teorema 3 vai nos auxiliar na demonstração da terminação do algoritmo de BTA. Vamos supor que o número de nomes de funções do alfabeto Υ_{in} da especificação seja finito. Na realidade, vamos precisar apenas de uma suposição mais fraca: o número de nomes de funções **referenciadas** nas regras de inicialização e transição deve ser finito. E essa suposição é desnecessária, pois é uma conseqüência do tamanho finito das regras de inicialização e transição, uma vez que os nomes de funções são objetos sintáticos que fazem parte das regras.

Theorem 4 . *Uma execução do algoritmo de BTA descrito pelas Figuras 5.3, 5.4, 5.5 e 5.6 sempre termina em um número finito de passos.*

Prova: Vamos supor que o algoritmo de BTA seja executado com um estado inicial S_0 onde `BSET` é $\{\texttt{init}, \texttt{prog}\}$. O algoritmo se resume a dois conjuntos de regras. O primeiro conjunto de regras é executado se o conjunto `BSET` é não vazio. O segundo, executado se `BSET` é vazio e `change` é *true*, torna `BSET` não vazio e altera o valor de `change` para *false*. Como mostramos que o Teorema 3 é válido, basta mostrar então que a função `change` recebe o valor *true* apenas um número finito de vezes. Isso é verificado pelos argumentos a seguir:

- A função `change` é inicializada com o valor *false*.
- Os únicos pontos em que `change` é alterada para *true* são encontrados nas Figuras 5.5 e 5.6.
- Essa alteração só é executada quando existe um nome de função f ao qual `bta_val` associa o valor *BTAPOS*. A classificação do nome de função f é alterado para *BTA-NEG*.
- De acordo com nossas suposições, existe um número finito de nomes de funções refereciadas na especificação. Assim, obviamente existe um número finito de nomes de funções inicialmente classificados como *BTAPOS*.
- Em nenhum ponto do algoritmo a classificação de um nome de função é alterada para *BTAPOS*.
- A guarda "`bta_val(`f`) =` *BTAPOS*" só será satisfeita um número finito de vezes, logo a função `change` só pode receber o valor *true* um número finito de vezes.

O primeiro conjunto de regras é sempre executado seqüencialmente em um número finito de passos. Essa seqüência pode alterar o valor de `change` para *true*, permitindo uma posterior execução do segundo conjunto de regras. Mas isso também só acontece um número finito de vezes, como mostramos acima. Logo o algoritmo irá sempre terminar em um número finito de passos. ■

Os teoremas desta seção nos mostram que o comportamento do algortimo de BTA da Seção 5.3 é bem simples e previsível. Os únicos tipos de estados que ocorrem durante uma execução desse algortimo são seqüências de estado $\langle S_p, \ldots, S_q \rangle, q > p$, em que `BSET` $= \{\texttt{init}, \texttt{prog}\}$ em S_p, e `BSET` $= \emptyset$ em S_q. No estado inicial, temos `BSET` $= \{\texttt{init}, \texttt{prog}\}$. Quando o conjunto `BSET` se torna vazio, a função `change` é testada e uma nova seqüência $\langle S_p, \ldots, S_q \rangle$ pode ser gerada, sucessivamente. Quando `change` for *false* no estado S_q, a execução termina.

6.2.2 Produção de uma Divisão Congruente

O objetivo do algoritmo de BTA, como explicado anteriormente, é classificar as funções de uma especificação ASM em estáticas (positivas) e dinâmicas (negativas). A classificação dada por um usuário às funções de entrada da especificação constitui o início desse processo.

As funções que dependem de uma função dinâmica devem ser também marcadas como dinâmicas. Essa dependência será formalizada a seguir. A classificação produz o que chamamos de *divisão congruente* das funções, isto é, uma divisão entre as categorias estática/dinâmica, que satisfaz os critérios da Definição 16.

Definition 16 *Em uma especificação ASM* $\mathcal{S} = (\Upsilon_{in}, \mathcal{D}, Init, Prog)$*, uma* divisão *das funções é dita* congruente *se satisfaz os seguintes requisitos:*

1. *Suponha que as regras Init ou Prog contenham uma atualização da forma*

$$f(t^*) \texttt{ := } g(u^*)$$

onde f *e* g *são nomes de funções e* t^* *e* u^* *são tuplas de termos cujo tamanho coincide com a aridade de* f *e* g*, respectivamente. Então se* g *é classificado como dinâmico (negativo),* f *também deve ser classificado como dinâmico.*

2. *Suponha que as regras Init ou Prog contenham um termo da forma*

$$f\ (g_1(t_1^*), \ldots, g_k(t_k^*))$$

onde f *é o nome de uma função com aridade* k*,* g_i *são também nomes de funções e cada* t_i^* *é uma tupla de termos cujo tamanho coincide com a aridade de* g_i*, para* $1 \leq i \leq k$*. Então se qualquer* $g_i, 1 \leq i \leq k$ *for classificado como dinâmico (negativo),* f *também deve ser classificado como dinâmico.*

Nesta seção, vamos procurar demonstrar que o algoritmo de BTA apresentado na Seção 5.3 constrói uma *divisão congruente* das funções de uma especificação ASM. A classificação, como discutido anteriormente, será dada pela função `bta_val`.

Definition 17 *Suponha que cada componente sintático TERM e RULE de uma regra ASM possa ser unicamente identificado. Por exemplo, duas regras de atualização que possuam exatamente o mesmo formato, localizadas em dois blocos diferentes, serão identificadas de modo diferente. Uma relação* $\sqsubseteq$ [1] *sobre objetos sintáticos TERM e RULE unicamente identificados é definida como a seguir, onde* $x, y, z \in$ *TERM, RULE:*

- $x \sqsubseteq block(\langle r_1, \ldots, r_k \rangle)$ *se, e somente se,* $x \sqsubseteq r_1$ *ou ...* *ou* $x \sqsubseteq r_k$.
- $x \sqsubseteq cond(t, r_1, r_2)$ *se, e somente se,* $x \sqsubseteq t$ *ou* $x \sqsubseteq r_1$ *ou* $x \sqsubseteq r_2$.
- $x \sqsubseteq update(t_1, t_2)$ *se, e somente se,* $x \sqsubseteq t_1$ *ou* $x \sqsubseteq t_2$.
- $x \sqsubseteq term(f, \langle t_1, \ldots, t_k \rangle)$ *se, e somente se,* $x \sqsubseteq t$ *ou ...* *ou* $x \sqsubseteq t_k$.
- $x \sqsubseteq x$.
- *Se* $x \sqsubseteq y$ *e* $y \sqsubseteq z$ *então* $x \sqsubseteq z$.

A necessidade de identificar os objetos sintáticos de modo único é importante para a demonstração dos teoremas desta seção. Observe que, se duas regras de atualização r_1 e r_2 possuem exatamente o mesmo formato, mas r_1 está inserida no bloco b_1 e r_2 está inserida no bloco b_2, então $r_1 \sqsubseteq b_1$ e $r_2 \sqsubseteq b_2$, mas não são válidas as relações $r_1 \sqsubseteq b_2$ e $r_2 \sqsubseteq b_1$. Se os objetos sintáticos não fossem identificados de modo único, as relações $r_1 \sqsubseteq b_2$ e $r_2 \sqsubseteq b_1$ também seriam válidas.

A definição que iremos apresentar a seguir caracteriza um determinado trecho de uma execução do algoritmo de BTA da Seção 5.3, e dá a esse trecho o nome de *ciclo de análise de componentes sintáticos.*

Definition 18 *Suponha uma seqüência de estados ASM* $\langle S_p, \ldots, S_q \rangle, q > p$, *representando um trecho de uma execução do algoritmo de BTA da Seção 5.3. Suponha que, em* S_p, *tenhamos* $\texttt{BSET} = \{\texttt{init}, \texttt{prog}\}$ *e, em* S_q, $\texttt{BSET} = \emptyset$, *onde* `init` *representa a regra de inicialização e* `prog` *a regra de transição da especificação ASM de entrada. Suponha ainda que* $\nexists S_k, p < k < q$, *onde* $\texttt{BSET} = \emptyset$, *ou seja,* S_q *é o primeiro estado após* S_p *em que o conjunto* BSET *é vazio. Vamos chamar o trecho da execução do algoritmo de BTA determinado por* $\langle S_p, \ldots, S_q \rangle$ *de* ciclo de análise dos componentes sintáticos *de uma especificação ASM.*

Observe que, de acordo com o Teorema 3, para cada estado como S_p descrito acima, um estado como S_q sempre existirá em uma execução do algoritmo de BTA. Ou seja, os ciclos de análise dos componentes sintáticos são sempre finitos.

Theorem 5 *Em um ciclo de análise dos componentes sintáticos, todas as regras de atualização da especificação ASM de entrada são processadas.*

[1] *Se* x *e* y *são dois componentes sintáticos,* $x \sqsubseteq y$ *lê-se* x faz parte de y.

Prova: Seja $\langle S_p, \ldots, S_q \rangle, q > p$, um ciclo de análise dos componentes sintáticos em uma execução do algoritmo de BTA. Suponha que $R_1 = update(t_1, t_2)$ seja uma regra de atualização qualquer da especificação de entrada, isto é, $R_1 \sqsubseteq$ `init` ou $R_1 \sqsubseteq$ `prog`. Para demonstrar o Teorema 5, basta provar que a seguinte afirmação será verdadeira, nos estados $S_k, p \leq k \leq q$:

Se	$\nexists R_2 \in$ `BSET` tal que $R_1 \sqsubseteq R_2$
então	$\exists S_l, p < l \leq k$, estado resultante da execução da regra da Figura 5.5 com a variável x instanciada como $x = R_1$.

Ou seja, se não existe nenhum elemento R_2 em `BSET` tal que $R_1 \sqsubseteq R_2$, então a atualização R_1 foi processada pelo algoritmo de BTA no passo anterior a um estado $S_l, l \leq k$.

A prova será realizada utilizando indução matemática sobre a seqüência de estados $S_k, p \leq k \leq q$.

- Em S_p, o estado inicial da seqüência, a afirmação

  $$\exists R_2 \in \texttt{BSET} \text{ tal que } R_1 \sqsubseteq R_2$$

  será obviamente verdadeira. Isso acontece porque supomos que, em S_p, `BSET` = {`init`, `prog`} e $R_1 \sqsubseteq$ `init` ou $R_1 \sqsubseteq$ `prog`.

- Vamos supor, como hipótese de indução, que o Teorema 5 é válido para um estado $S_k, p \leq k < q$:

  - Se a afirmação

    $$\nexists R_2 \in \texttt{BSET} \text{ tal que } R_1 \sqsubseteq R_2$$

    for verdadeira em S_k, então, pela hipótese de indução, $\exists S_l, p < l \leq k$, estado resultante da execução da regra da Figura 5.5 com a variável x instanciada como $x = R_1$. Se $l \leq k$, então também podemos afirmar que $l \leq k + 1$. No estado S_{k+1}, existirá um estado S_l que satisfaz o Teorema 5, independente do fato de que exista ou não um elemento R_2 em `BSET` tal que $R_1 \sqsubseteq R_2$.

  - Se a afirmação

    $$\exists R_2 \in \texttt{BSET} \text{ tal que } R_1 \sqsubseteq R_2 \tag{6.1}$$

    for verdadeira em S_k, então temos que verificar todos casos possíveis para a construção de um novo estado S_{k+1}. Uma situação que mantenha a afirmação 6.1 verdadeira significa que o Teorema 5 continua válido. Se uma situação fizer com que essa afirmação se torne falsa, então devemos analisar se um estado S_l como descrito no Teorema 5 foi produzido. O algoritmo escolhe um elemento x em `BSET`, que será processado pela macro `ProcessBTA`:

    * Se $x = block(\langle r_1, \ldots, r_k \rangle)$, duas são as possibilidades. Na primeira, $R_1 \not\sqsubseteq x$. Então retirar x de `BSET` não modifica a afirmação 6.1 acima, assim o Teorema 5 também será válido para o estado S_{k+1}. A segunda possibilidade ocorre quando $R_1 \sqsubseteq x$. Usando a Definição 17, podemos afirmar que $\exists r_i$ tal que $R_1 \sqsubseteq r_i$. Mas a retirada de x foi compensada com a inserção de r_i em `BSET`. Novamente, o Teorema 5 também será válido para o estado S_{k+1}.

* Se $x = cond(c, r_1, r_2)$, podemos seguir um raciocínio equivalente ao aplicado para o caso em que x é um bloco de regras. Neste caso, devemos verificar se $R_1 \sqsubseteq r_1$ ou $R_1 \sqsubseteq r_2$
* Se $x = term(f, t^*)$, podemos afirmar que $R_1 \not\sqsubseteq x$, logo a afirmação 6.1 acima continua verdadeira e o Teorema 5 também será válido para o estado S_{k+1}.
* Se $x = update(t_3, t_4)$, duas são as possibilidades. Se $R_1 \neq x$, retirar x de `BSET` não modifica a afirmação 6.1 acima. Se $R_1 = x$, a afirmação 6.1 poderá se tornar falsa (na realidade, sempre se tornará falsa). Mas, nesse caso, o estado S_{k+1} será exatamente o estado S_l a que se refere o Teorema 5.

Desse modo, mostramos que o Teorema 5 é válido para qualquer estado da seqüência $\langle S_p, \ldots, S_q \rangle, q > p$. ■

O teorema a seguir apresenta uma condição parcial para que o algoritmo de BTA produza uma divisão congruente.

Lemma 6 *Na execução do algoritmo de BTA, suponha um estado em que exista alguma regra de atualização que viole o primeiro princípio para uma divisão congruente das funções da especificação de entrada. Então esse não será o último estado dessa execução, ou seja, a execução não termina nesse estado.*

Prova: O corpo principal do algoritmo de BTA é exibido na Figura 5.3. Sua execução termina somente se o conjunto `BSET` é vazio e `change` é *false*. O conjunto `BSET` é vazio apenas no final de uma seqüência de estados que convencionamos chamar de ciclo de análise de componentes sintáticos. Para provar o teorema basta, então, mostrar que `change` é sempre *false* nesses estados, se o primeiro princípio da congruência é violado.

De uma maneira mais formal, suponha que $\langle S_p, \ldots, S_q \rangle, q > p$ seja um ciclo de análise de componentes sintáticos. Suponha que as regras `init` ou `prog` contenham uma atualização R da forma

$$f(t^*) \texttt{ := } g(u^*)$$

ou seja, $R \sqsubseteq$ `init` ou $R \sqsubseteq$ `prog`, onde f e g são nomes de funções, e t^* e u^* são tuplas de termos cujo tamanho coincide com a aridade de f e g, respectivamente. No estado S_q, se $\texttt{bta\_val}(g) = BTANEG$ e $\texttt{bta\_val}(f) = BTAPOS$ então `change` deve ser *true*.

Se $\texttt{bta\_val}(f) = BTAPOS$ no estado S_q, então $\texttt{bta\_val}(f)$ também é $BTAPOS$ no estado S_p. Isso acontece porque os únicos valores possíveis para a imagem da função `bta_val` são $BTAPOS$ e $BTANEG$ e não existe nenhum ponto no algoritmo alterando a classificação de um nome de função de $BTANEG$ para $BTAPOS$.

Por outro lado, se $\texttt{bta\_val}(g) = BTANEG$ no estado S_q, então $\texttt{bta\_val}(g)$ poderia ser $BTAPOS$ ou $BTANEG$ no estado S_p. Vamos analisar os dois casos:

1. Suponha que $\texttt{bta\_val}(g) = BTANEG$ no estado S_p. O Teorema 5 mostra que **todas** as regras de atualizações r tal que $r \sqsubseteq$ `init` ou $r \sqsubseteq$ `prog` são processadas pelo algoritmo de BTA em uma seqüência de estados como a apresentada. Assim, ao processar R, o valor de $\texttt{bta\_val}(f) = BTAPOS$ teria sido alterado para $BTANEG$, de acordo com a regra da Figura 5.5. Logo essa hipótese é impossível.

2. Suponha que $\texttt{bta\_val}(g) = BTAPOS$ no estado S_p. Em apenas dois pontos do algoritmo de BTA a função `bta_val` recebe o valor *BTANEG*. Esses pontos ocorrem nas Figuras 5.5 e 5.6, mas em ambos a função `change` é necessariamente atualizada com o valor *true*.

Assim podemos afirmar que $\texttt{change} = true$ no estado S_q. ■

O teorema a seguir é similar ao Teorema 5, só que nesse caso trata de termos, em vez de regras de atualização.

Lemma 7 *Em um ciclo de análise dos componentes sintáticos, todos os termos presentes em expressões da especificação ASM de entrada são processados.*

Prova: Seja $\langle S_p, \ldots, S_q \rangle, q > p$, um ciclo de análise dos componentes sintáticos em uma execução do algoritmo de BTA. Suponha que $T = term(f, t^*)$, onde $T \sqsubseteq \texttt{init}$ ou $T \sqsubseteq \texttt{prog}$. Para demonstrar o Teorema 7, basta provar que a seguinte afirmação será verdadeira, nos estados $S_k, p \leq k \leq q$:

Se	$\nexists X \in \texttt{BSET}$ tal que $T \sqsubseteq X$
então	$\exists S_l, p < l \leq k$, estado resultante da execução da regra da Figura 5.6 com a variável x instanciada como $x = T$.

Ou seja, se não existe nenhum elemento X em `BSET` tal que $T \sqsubseteq X$, então o termo T foi processado pelo algoritmo de BTA no passo anterior a um estado $S_l, l \leq k$.

A prova será realizada utilizando indução matemática sobre a seqüência de estados $S_k, p \leq k \leq q$.

- Em S_p, o estado inicial da seqüência, a afirmação

$$\exists X \in \texttt{BSET} \text{ tal que } T \sqsubseteq X$$

será obviamente verdadeira. Isso acontece porque supomos que, em S_p, $\texttt{BSET} = \{\texttt{init}, \texttt{prog}\}$ e $X \sqsubseteq \texttt{init}$ ou $X \sqsubseteq \texttt{prog}$.

- Vamos supor, como hipótese de indução, que o Teorema 7 é válido para um estado $S_k, p \leq k < q$:

  - Se a afirmação

$$\nexists X \in \texttt{BSET} \text{ tal que } T \sqsubseteq X$$

for verdadeira em S_k, então, pela hipótese de indução, $\exists S_l, p < l \leq k$, estado resultante da execução da regra da Figura 5.6 com a variável x instanciada como $x = T$. Se $l \leq k$, então também podemos afirmar que $l \leq k + 1$. No estado S_{k+1}, existirá um estado S_l que satisfaz o Teorema 7, independente do fato de que exista ou não um elemento X em `BSET` tal que $T \sqsubseteq X$.

  - Se a afirmação

$$\exists X \in \texttt{BSET} \text{ tal que } T \sqsubseteq X \tag{6.2}$$

for verdadeira em S_k, então temos que verificar todos casos possíveis para a construção de um novo estado S_{k+1}. Um elemento x será escolhido em `BSET` e processado pela macro `ProcessBTA`:

* Se $x = block(\langle r_1, \ldots, r_k \rangle)$, duas são as possibilidades. Na primeira, $T \not\sqsubseteq x$. Então retirar x de `BSET` não modifica a afirmação 6.2 acima, assim o Teorema 7 também será válido para o estado S_{k+1}. A segunda possibilidade ocorre quando $T \sqsubseteq x$. Usando a Definição 17, podemos afirmar que $\exists r_i$ tal que $R_1 \sqsubseteq r_i$. Mas a retirada de x foi compensada com a inserção de r_i em `BSET`. Novamente, o Teorema 7 também será válido para o estado S_{k+1}.
* Se $x = cond(c, r_1, r_2)$, podemos seguir um raciocínio equivalente ao aplicado para o caso em que x é um bloco de regras. Neste caso, devemos verificar se $T \sqsubseteq c$, $T \sqsubseteq r_1$ ou $T \sqsubseteq r_2$.
* Se $x = update(t_1, t_2)$, um procedimento semelhante ao adotado para blocos de regras e condicionais pode ser realizado. Neste caso, devemos verificar se $T_1 \sqsubseteq t_1$ ou $T \sqsubseteq t_2$.
* Se $x = term(f, t^*)$, duas são as possibilidades. Se $T \neq x$, retirar x de `BSET` não modifica a afirmação 6.2 acima. Se $T = x$, a afirmação 6.2 poderá se tornar falsa (na realidade, sempre se tornará falsa). Mas, nesse caso, o estado S_{k+1} será exatamente o estado S_l a que se refere o Teorema 7.

Desse modo, mostramos que o Teorema 7 é válido para qualquer estado da seqüência $\langle S_p, \ldots, S_q \rangle, q > p$. ■

O teorema a seguir apresenta mais uma condição parcial para que o algoritmo de BTA produza uma divisão congruente.

Lemma 8 *Na execução do algoritmo de BTA, suponha um estado em que exista algum termo que viole o segundo princípio para uma divisão congruente das funções da especificação de entrada. Então esse não será o último estado dessa execução, ou seja, a execução não termina nesse estado.*

Prova: De maneira semelhante à apresentada na prova do Teorema 6, para demonstrar o Teorema 8, basta mostrar que `change` é sempre *false* no final de um ciclo de análise de componentes sintáticos, se o segundo princípio da congruência é violado.

De uma maneira mais formal, suponha que $\langle S_p, \ldots, S_q \rangle, q > p$ seja um ciclo de análise de componentes sintáticos. Suponha que as regras `init` ou `prog` contenham um termo T da forma

$$f\ (g_1(t_1^*), \ldots, g_k(t_k^*))$$

ou seja, $T \sqsubseteq$ `init` ou $T \sqsubseteq$ `prog`, onde f é o nome de uma função com aridade k, g_i são também nomes de funções e cada t_i^* é uma tupla de termos cujo tamanho coincide com a aridade de g_i, para $1 \leq i \leq k$. No estado S_q, se `bta_val`$(g_j) = BTANEG$, para algum $1 \leq j \leq k$, e `bta_val`$(f) = BTAPOS$ então `change` deve ser *true*.

Se $\exists g_j, 1 \leq j \leq k$, tal que `bta_val`$(g_j) = BTANEG$ no estado S_q, então `bta_val`(g_j) poderia ser $BTAPOS$ ou $BTANEG$ no estado S_p. Vamos analisar os dois casos:

1. Suponha que `bta_val`$(g_j) = BTANEG$ no estado S_p. O Teorema 7 mostra que, na seqüência de estados analisada, **todos** os termos t tal que $t \sqsubseteq$ `init` ou $t \sqsubseteq$ `prog` são processados pelo algoritmo de BTA. Assim, ao processar T, o valor de `bta_val`$(f) =$

BTAPOS teria sido alterado para *BTANEG*, de acordo com a regra da Figura 5.6. Logo essa hipótese é impossível.

2. Suponha que $\texttt{bta\_val}(g_j) = BTAPOS$ no estado S_p. Em apenas dois pontos do algoritmo de BTA a função `bta_val` recebe o valor *BTANEG*. Esses pontos ocorrem nas Figuras 5.5 e 5.6, mas em ambos a função `change` é necessariamente atualizada com o valor *true*.

Assim podemos afirmar que `change` = *true* no estado S_q. ■

Os Teoremas 6 e 8 serão usados para demonstrar que o algoritmo de BTA da Seção 5.3 produz uma divisão congruente.

Theorem 9 *O algoritmo de BTA da Seção 5.3 produz uma divisão congruente das funções de uma especificação ASM.*

Prova: Com o Teorema 3, vimos que uma execução do algoritmo de BTA consiste em seqüências de estado $\langle S_p, \ldots, S_q \rangle, q > p$, que se repetem, no caso da função `change` tiver o valor *true*. No estado S_p, $\texttt{BSET} = \{\texttt{init}, \texttt{prog}\}$ e S_q é o primeiro estado após S_p em que $\texttt{BSET} = \emptyset$. O Teorema 4 provou que o algoritmo de BTA sempre termina. Os Teoremas 6 e 8 mostraram que, em um estado em que $\texttt{BSET} = \emptyset$, o valor de `change` será *true* se existir alguma condição que viole os critérios de congruência expressos na Definição 16. Isso impede que o algoritmo termine. Mas como sabemos que o algoritmo irá terminar em um número finito de passos (`change` será *false*), os critérios de uma divisão congruente serão satisfeitos em um número finito de passos. ■

Com isso atingimos o objetivo desta seção, que era provar que o algoritmo de BTA da Seção 5.3 produz uma divisão congruente. Não vamos desenvolver nenhuma prova sobre o algoritmo que produz a especificação anotada, embora essa especificação anotada seja utilizada na seção seguinte. Vamos supor que o algoritmo da Figura 5.7 esteja correto, produzindo anotações dinâmicas para termos dinâmicos, estáticas para termos estáticos e construtores *lift* para termos estáticos inseridos em contextos dinâmicos.

6.3 Especialização

Na Seção 5.4, um algoritmo para especialização de especificações ASM anotadas é apresentado. Nesta seção, vamos discutir e provar algumas propriedades importantes sobre esse algoritmo. Vamos discutir as condições de terminação do algoritmo, as condições de terminação dos programas residuais gerados e também a correta semântica desses programas residuais.

6.3.1 Critério de Correção para a Especialização

Suponha P uma especificação ASM com duas entradas que, quando aplicada às entradas in_1 e in_2, produz a saída *out*. Suponha que P^{an} seja uma versão anotada dessa especificação, usando o algoritmo da Seção 5.3, com a entrada in_1 estática e a entrada in_2

dinâmica. Suponha também que *Spec* represente o algoritmo de especialização apresentado na Seção 5.4. As seguintes equações podem ser usadas para determinar a correção de *Spec*:

$$\begin{aligned} out &= [\![P]\!]\,(in_1, in_2) \\ P^{an} &= [\![BTA]\!]\,(P,\ \{in_1 \approx S,\ in_2 \approx D\}) \\ P_{in_1} &= [\![Spec]\!]\,(P^{an}, in_1) \\ out &= [\![P_{in_1}]\!]\,(in_2) \end{aligned}$$

A especificação P^{an} é a *especificação de entrada* para o algoritomo de especialização. A especificação P_{in_1} é a *especificação residual* gerada pelo especializador, em relação à entrada in_1. Essas equações podem ser facilmente generalizadas para um número arbitrário de entradas.

Observe que a especificação residual, quando executada com a entrada dinâmica in_2, produz a mesma saída que a especificação original, quando executada sobre ambas as entradas. Ou seja, nas condições estabelecidas, P_{in_1} possui a mesma semântica que P. Para verificar a correção do algortimo de especificação, portanto, devemos comparar as saídas produzidas por P e P_{in_1}. Vamos convencionar que todas as saídas de uma especificação ASM são produzidas por funções externas.

Vamos supor que Υ e Υ^{res} sejam os vocabulários associados a P e P_{in_1}, cujas álgebras chamaremos de $\mathcal{A}$ e $\mathcal{A}^{res}$, respectivamente. No Capítulo 5, designamos o vocabulário da especificação de entrada P por Υ_{in}, agora usaremos Υ para não sugerir nenhuma relação com as entradas de P. Como estabelecido na Seção 5.4.7, todos os nomes de funções de Υ^{res} pertencem também a Υ, com exceção de uma função adicional `curstate` : *INT*, que define o fluxo de controle na especificação residual. Por enquanto, vamos supor que o algoritmo de especialização *Spec* termine em um número finito de passos (caso contrário não teria produzido a especificação P_{in_1}), e que seu vocabulário será Υ^{Spec}. Vamos considerar a algébra $\mathcal{A}^{Spec}$, associada a *Spec*, após a execução do último passo do algoritmo, ou seja, o valor final das funções.

Na Seção 5.4.7, descrevemos informalmente como seria o formato da regra de transição da especificação residual gerada, baseada nos valores armazenados na função `gencode`, após o último passo da execução de *Spec*. Na Seção 6.3.2 vamos descrever formalmente como será essa regra de transição, mas por enquanto vamos nos basear na descrição informal.

Para provar que P_{in_1} executado sobre a entrada in_2 possui a mesma semântica que P executado sobre ambas as entradas, uma das propriedades que vamos mostrar é que as funções de $\Upsilon \cap \Upsilon^{res}$ (funções dinâmicas de P) possuem exatamente os mesmos valores, em cada passo k da execução de P e P_{in_1}. Na realidade, isso não é condição necessária nem suficiente para demonstrar a equivalência da semântica, assim outros aspectos devem ser analisados. Não é condição necessária porque P_{in_1} poderia produzir exatamente a mesma saída que P, mas sem necessariamente ter os mesmos valores para as funções em $\Upsilon \cap \Upsilon^{res}$ exatamente nos **mesmos passos** da execução. A forma como o algoritmo de especialização gera a especificação residual, sem transição de compressões, irá nos ajudar a provar essa propriedade, uma vez que cada passo da especificação P terá um correspondente em P_{in_1}. Também não é condição suficiente para demonstrar a equivalência da semântica, porque a saída de P_{in_1} é produzida por funções externas envolvendo parâmetros dinâmicos e também

```
f1: INT → RULE

f1(n) ≡
  if n < TotS then
    cond (term ("=", ⟨term("curstate", null), lift(n)⟩),
        f2 (n, gencode(⟨n, 0⟩)),
        f1 (n + 1) )
  else
    block(null)
  endif
```

Figura 6.2: Função `f1`.

valores construídos com as entradas estáticas. Assim devemos analisar não só os valores dinâmicos, associados às funções de $\Upsilon \cap \Upsilon^{res}$, mas considerar também os valores estáticos de P^{an} que são usados na construção da especificação P_{in_1}. Sabendo que as funções dinâmicas e estáticas possuem interpretações equivalentes em cada passo de P e P_{in_1}, vamos mostrar que chamadas de funções externas são executadas, com os mesmos valores, em cada passo da execução de ambas as especificações. Essa abordagem será utilizada na Seção 6.3.3.

Observe que, com uma abordagem semelhante à descrita acima, não é necessário que as especificações terminem em um número finito de passos para que a sua semântica seja considerada equivalente.

6.3.2 Construindo a Especificação Residual

Antes de abordar os aspectos relacionados à equivalência de semântica entre uma especificação de entrada e uma especificação residual gerada a partir da mesma, vamos definir formalmente o formato das especificações residuais. O processo foi descrito informalmente na Seção 5.4.7.

A regra de inicialização de uma especificação residual terá sempre o formato

```
curstate := 0
```

onde `curstate` : *INT* é uma função que não pertence a Υ.

A regra de transição será constituída por uma seqüência de regras condicionais aninhadas, testando o valor de `curstate`. O formato exato da regra de transição é obtido com a chamada de função `f1`(0), onde `f1` é a função apresentada na Figura 6.2. Essa função consulta os valores de `Tots` e `gencode`, definidas no especializador, e utiliza uma segunda função `f2`, apresentada na Figura 6.3.

O algoritmo de especialização poderia executar um passo adicional, após o término de sua execução normal, onde geraria a especificação residual. Uma solução interessante seria usar uma chamada a uma função externa com o valor de `f1`(0). Essa função externa poderia, por exemplo, gerar o texto da especificação residual em um arquivo.

```
f2: INT × GRULE → RULE

f2 (s, update(t1, t2)) ≡
  update(t1, t2)
f2 (s, gcond(t, j, k)) ≡
  cond (t, f2 (s, gencode(⟨s, j⟩)), f2 (s, gencode(⟨s, k⟩)))
f2 (s, gblock(j, k)) ≡
  merge_blocks (f2 (s, gencode(⟨s, j⟩)), f2 (s, gencode(⟨s, k⟩)))
```

Figura 6.3: Função `f2`.

6.3.3 Correção do Algoritmo de Especialização

Vamos considerar três especificações ASM concomitantemente:

1. Uma especificação P com duas entradas, executada sobre in_1 e in_2. Essa será a *especificação de entrada* para o avaliador parcial. Seu vocabulário será designado por Υ.

2. O algoritmo de especialização *Spec*, executado sobre uma versão anotada P^{an} de P e a entrada in_1. A especificação anotada P^{an} é gerada pelo algoritmo *BTA* usando in_1 como entrada estática. O vocabulário de *Spec* é designado por Υ^{Spec}.

3. A especificação residual P_{in_1} gerada por Spec, que será executada sobre in_2. Seu vocabulário é Υ^{res}.

Vamos supor que *Spec* foi executado e terminou em um número finito de passos, produzindo P_{in_1}. Vamos analisar e comparar a execução de P e P_{in_1}.

O algoritmo BTA, quando executado sobre P, produz uma divisão das funções da especificação de entrada em estáticas e dinâmicas. Vamos chamar o conjunto dos nomes de funções estáticas de Υ^+ e o conjunto dos nomes de funções dinâmicas de Υ^-. É claro que $\Upsilon = \Upsilon^+ \cup \Upsilon^-$.

Suponha que $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots$ seja uma seqüência de estados produzida pela execução de P sobre in_1 e in_2. Cada estado é uma álgebra que pode ser dividida em dois conjuntos disjuntos, usando para isso os nomes de funções de Υ^+ e Υ^-. Para $i = 0, 1, 2, \ldots$, vamos chamar essas álgebras de $\mathcal{A}_i^+$ e $\mathcal{A}_i^-$, respectivamente. Observe que $\mathcal{A}_i = \mathcal{A}_i^+ \cup \mathcal{A}_i^-$.

De forma semelhante à acima, vamos supor que $\mathcal{A}_0^{res}, \mathcal{A}_1^{res}, \mathcal{A}_2^{res}, \ldots$ seja uma seqüência de estados produzida pela execução de P_{in_1} sobre in_2. Vamos supor também que $\mathcal{A}^{Spec}$ seja o estado final da execução do especializador *Spec*.

Como vamos analisar nomes de funções de três especificações ao mesmo tempo, vamos usar a notação $f^{\mathcal{B}}$ para indicar que estamos nos referindo à interpretação da função f na álgebra $\mathcal{B}$.

Theorem 10 *Se a execução do algoritmo de especialização da Seção 5.4 terminar em um número finito de passos, a especificação residual produzida será sempre correta.*

Prova: Seja $k \geq 0$ um número inteiro para o qual exista um estado $\mathcal{A}_k$ na execução de P sobre in_1 e in_2, e um estado $\mathcal{A}^{res}_{k+1}$ na execução de P_{in_1} sobre in_2. Para demonstrar o Teorema 10 vamos provar que:

1. $\mathcal{A}^{-}_{k} = \mathcal{A}^{res}_{k+1} - \{\texttt{curstate}^{\mathcal{A}^{res}_{k+1}}\}$.
2. $f^{\mathcal{A}_k}(t^*) = \texttt{TSpec}^{\mathcal{A}^{Spec}}(\langle f, t^*\rangle, \texttt{curstate}^{\mathcal{A}^{res}_{k+1}})$, para todo $f \in \Upsilon^{+}$.
3. Se uma chamada de função externa $f(t_1, \ldots, t_j)$ é executada no passo k de P, então será executada também no passo $k+1$ de P_{in_1} e todos os termos avaliados serão equivalentes, isto é, $t_i^{\mathcal{A}_k} = t_i^{\mathcal{A}^{res}_{k+1}}$, para $i = 1, \ldots, j$.

A afirmação 1 diz que as funções da especificação residual P_{in_1} possuem os mesmos valores que as funções de P classificadas como dinâmicas, em cada passo da execução das duas especificações ASM. A exceção é a função de nome `curstate`, que não está presente em Υ^{-}. Observe que essa equivalência de valores inicia com o estado $\mathcal{A}^{-}_{0}$ de P e $\mathcal{A}^{res}_{1}$ de P_{in_1}.

A afirmação 2 diz que o valor de todas as funções estáticas de P, no k-ésimo passo de sua execução, são representadas de forma exata na tabela `TSpec` do especializador. Essa representação é encontrada na "posição" `curstate` de `TSpec`, usando o valor da interpretação de `curstate` no $(k+1)$-ésimo passo da execução de P_{in_1}. Durante a demonstração do teorema, quando nos referirmos a uma *posição* j da tabela `TSpec`, estaremos considerando os valores $\texttt{TSpec}(l, j)$ para todo $l \in \texttt{Dom(1,TSpec)}$.

As duas primeiras afirmativas são na realidade uma forma de se incrementar o invariante, de modo a facilitar a demonstração do teorema. A afirmação 3 é a que define a equivalência semântica entre as especificações P e P_{in_1}. O que chamamos de passo k da execução de P é o disparo das regras de transição sobre o estado $\mathcal{A}_k$, gerando um estado $\mathcal{A}_{k+1}$. Uma observação importante, que será usada na demonstração a seguir, é que todas as chamadas de funções externas são classificadas como dinâmicas na especificação anotada P^{an}.

A prova será realizada usando indução matemática sobre k, o número de passos da execução de P e P_{in_1}. Algumas propriedades do funcionamento interno do especializador serão assumidas como verdadeiras, usando apenas explicações informais. Vamos nos concentrar na interação entre as três ASM: P, *Spec* e P_{in_1}.

Primeira Parte - Caso Base:

Primeiro vamos analisar o caso em que $k = 0$. Para determinar os valores das funções de $\mathcal{A}^{res}_{k+1}$, ou seja, $\mathcal{A}^{res}_{1}$, devemos analisar como a regra de transição de P_{in_1} foi construída.

Na Seção 6.3.2, definimos que a regra de inicialização da especificação residual P_{in_1} faz $\texttt{curstate}^{\mathcal{A}^{res}_{0}} = 0$. A regra de transição residual é uma seqüência de testes sobre o valor da função `curstate`, assim é fácil ver que $\mathcal{A}^{res}_{1}$ é um estado construído pela execução de $\texttt{f2}(0, \texttt{gencode}(\langle 0, 0\rangle))$. Devemos analisar então como essa regra residual foi construída pelo especializador.

No especializador *Spec*, algumas das inicializações importantes são

$$\texttt{CurS} = 0, \ \texttt{TotS} = 0, \ \texttt{BuildNewState} = 0, \ \texttt{GSET} = \{\langle \texttt{init}, 0, \mathit{null}\rangle\}$$

O valor $\texttt{init}^{\mathcal{A}_0^{res}}$, nesse caso, representa a regra de inicialização da especificação de entrada para P_{in_1}, ou seja, a regra de inicialização de P^{an}. Como vamos nos referir aos valores das funções de Spec em diversos passos de sua execução, vamos omitir os índices que denotam a álgebra associada, mas deixando claro os valores com que estamos lidando.

Observando o código do algoritmo de especialização apresentado na Figura 5.14, podemos ver que as regras designadas por `SPECRULES` serão disparadas no primeiro passo da execução. Essas regras são apresentadas na Figura 5.10.

No Capítulo 5, convencionamos que a regra de inicialização de uma especificação ASM é representada por um bloco de regras de atualização. Assim, `GSET` inicialmente conterá um elemento representando um bloco de regras de atualização (regra de inicialização de P^{an}), associado a um número de código 0 e um conjunto vazio de atualizações estáticas coletadas. A condição da regra do lado esquerdo da Figura 5.10 é satisfeita enquanto `GSET` for não vazio. Cada atualização do bloco será processada, de acordo com o algoritmo descrito pelas regras das Figuras 5.11 (blocos) e 5.12 (atualizações).

Se a primeira atualização do bloco é estática, uma atualização é coletada no novo elemento inserido em `GSET`. Se a primeira atualização do bloco é dinâmica, uma atualização residual é gerada, com todas as informações estáticas computadas por *Reduce*. Vamos supor que a função *Reduce* funcione corretamente, e também que as regras residuais sejam corretamente geradas.

É fácil ver que o bloco inserido em `GSET` se torna menor a cada passo, garantindo que será vazio em um número finito de passos. Quando isso acontece, as atualizações coletadas são disparadas em paralelo e registradas em `VFuncs`. Vamos supor que a função *Val* funcione corretamente. Assim teremos, para toda atualização $f(t_1, \ldots, t_k)$ `:=` v pertencente a `init`, com $f \in \Upsilon^+$:

$$\texttt{VFuncs}(\langle f, \langle \mathit{Val}(t_1), \ldots, \mathit{Val}(t_k)\rangle\rangle) = \mathit{Val}(v)$$

Os passos a seguir constituem uma execução das regras da Figura 5.14. Como nesse momento `Dom(2,TSpec)` é vazio, a execução se torna trivial. O valor de `TotS` se torna 1, e a tabela `TSpec`, na posição 1, recebe os valores de `VFuncs`. Assim, para todo $l \in \texttt{Dom}(1, \texttt{VFuncs})$:

$$\texttt{TSpec}(l, 1) = \texttt{VFuncs}(l)$$

O código residual gerado por "`GenNextStep(NewState)`" será

```
curstate := 1
```

já que `NewState` será interpretado como o valor inteiro 1 (o mesmo valor que `TotS`). O valor de `BuildNewState` se torna novamente 0, finalizando a seqüência de processamento da regra de inicialização de P^{an}. Vamos analisar então os resultados nesse momento.

O bloco residual gerado é formado por versões reduzidas de todas as atualizações $f(t_1, \ldots, t_k)$ `:=` v pertencentes a `init`, com $f \in \Upsilon^-$. As informações estáticas foram computadas usando os valores de `VFuncs` (antes do disparo das atualizações estáticas coletas) e os valores das funções de entrada. O valor de `VFuncs`, nesse caso, era *undef* para todas as funções, em todos os endereços. Como $\texttt{curstate}^{\mathcal{A}_0^{res}} = 0$, é óbvio que esse bloco

residual será disparado no passo 1 da execução de P_{in_1}. Assim teremos:

$$\begin{aligned} \mathcal{A}_1^{res} - \{\texttt{curstate}^{\mathcal{A}_1^{res}}\} &= \mathcal{A}_0^- \\ \texttt{curstate}^{\mathcal{A}_1^{res}} &= 1 \end{aligned}$$

Como discutimos anteriormente, qualquer chamada de função externa de P é classificada como dinâmica na especificação anotada. Assim todas as chamadas de funções externas da regra de inicialização de P farão parte das versões reduzidas das atualizações geradas. Como supusemos que as funções *Reduce* e *Val* funcionam corretamente, podemos afirmar que, se uma chamada de função externa é executada no passo 0 de P, então será executada também no passo 1 de P_{in_1} e todos os termos avaliados serão equivalentes.

As atualizações estáticas de init foram todas coletadas e armazenadas na tabela *TSpec*, na posição 1. Sabendo que $\texttt{curstate}^{\mathcal{A}_1^{res}} = 1$, então

$$\texttt{TSpec}(\langle f, t^* \rangle, \texttt{1}) = \texttt{TSpec}(\langle f, t^* \rangle, \texttt{curstate}^{\mathcal{A}_1^{res}}) = f^{\mathcal{A}_0}(t^*), \text{ para todo } f \in \Upsilon^+$$

Assumindo que os valores de `TSpec` na posição 1 não são alterados nos passos seguintes da execução de *Spec*, então podemos afirmar que o Teorema 10 é válido quando $k = 0$.

Segunda Parte - Hipótese de Indução:

Supondo que o Teorema 10 seja válido para o passo k da execução da especificação P (passo $k+1$ de P_{in_1}), vamos provar que será válido também para o passo $k+1$ de P (passo $k+2$ de P_{in_1}).

Usando a hipótese de indução, podemos afirmar que o valor das funções dinâmicas de P no passo k é o mesmo que o observado em P_{in_1} no passo $k+1$. O valor das funções estáticas de P está armazenado na tabela `TSpec` do especializador, na posição $\texttt{curstate}^{\mathcal{A}_{k+1}^{res}}$.

Observando o formato da regra de transição residual, definido pela formalização apresentada na Seção 6.3.2, é fácil ver que $\mathcal{A}_{k+2}^{res}$ é um estado construído pela execução de $\texttt{f2}(\texttt{curstate}^{\mathcal{A}_{k+1}^{res}}, \texttt{gencode}(\langle \texttt{curstate}^{\mathcal{A}_{k+1}^{res}}, 0 \rangle))$. Devemos analisar então como essa regra residual foi construída pelo especializador.

Vamos analisar o algoritmo *Spec* a partir do ponto em que inicia a construção da regra residual em questão, com a execução das regras apresentadas no lado direito da Figura 5.10. Observe que, nesse caso, o valor de `CurS`+1 deverá ser o mesmo que $\texttt{curstate}^{\mathcal{A}_{k+1}^{res}}$, de modo que no passo seguinte de Spec a regra residual associada a $\texttt{curstate}^{\mathcal{A}_{k+1}^{res}}$ seja gerada.

O conjunto `GSET` será atualizado com um elemento representando `prog` (regra de transição de P^{an}), associado a um número de código 0 e um conjunto vazio de atualizações estáticas coletadas. A função `VFuncs` é atualizada com os valores da tabela `TSpec` na posição $\texttt{curstate}^{\mathcal{A}_{k+1}^{res}}$. Usando a hipótese de indução, isso quer dizer que `VFuncs` conterá exatamente o valor das funções estáticas de P no passo k, isto é, $f^{\mathcal{A}_k}$, para todo $f \in \Upsilon^+$.

Nos passos seguintes, a condição da regra do lado esquerdo da Figura 5.10 é satisfeita enquanto `GSET` for não vazio. Cada subregra da especificação anotada será processada, de acordo com o algoritmo descrito pelas regras das Figuras 5.11 (blocos), 5.12 (atualizações) e 5.13 (regras condicionais). É possível demonstrar que o conjunto `GSET` se torna vazio em um número finito de passos, se aplicarmos técnicas semelhantes às usadas na Seção 6.2.1.

Se a regra sendo analisada é uma regra condicional anotada, o algoritmo para seu processamento é exibido na Figura 5.13. Se a anotação associada indica que é uma regra

estática, a condição é avaliada pela função *Val*, que utiliza os valores de `VFuncs`. Como supusemos que *Val* funciona corretamente e `VFuncs` contém exatamente os valores das funções estáticas de P no passo k, o resultado do teste em *Spec* é o mesmo que em P. Ou seja, a regra residual que será executada no passo $k+1$ de P_{in_1} foi gerada obedecendo o mesmo fluxo de controle que o observado em P, no passo k. Analisando o segundo caso, se a anotação associada indica que é uma regra condicional dinâmica, uma regra condicional residual é gerada, com todas as informações estáticas computadas por *Reduce*, que também supusemos funcionar corretamente. Um fato importante para o correto funcionamento, a exemplo do que acontece quando *Val* é usada, é que `VFuncs` contém exatamente os valores das funções estáticas de P, no passo k. Dois novos elementos são inseridos em `GSET`, para processar as duas cláusulas da regra condicional. Quando a regra residual de P_{in_1} for executada no passo $k+1$, a cláusula escolhida será a mesma que em P, no passo k.

Outro ponto importante, relacionado à afirmação 3 deste teorema, é que as funções externas eventualmente utilizadas no teste da regra condicional anotada como dinâmica em P^{an} serão chamadas com os mesmos valores tanto em P, no passo k, quanto em P_{in_1}, no passo $k+1$. A justificativa para isso é a mesma dada para o fato da avaliação da condição dar o mesmo resultado nas duas especificações, ou seja, os valores das funções estáticas e dinâmicas possuem as mesmas interpretações nas duas especificações.

Se a regra sendo analisada é um bloco de regras anotadas, a primeira atualização do bloco é processada, de maneira equivalente ao descrito na primeira parte desta demonstração. Se a primeira atualização do bloco é estática, uma atualização é coletada e o elemento é novamente inserido em `GSET`. Se a primeira atualização do bloco é dinâmica, uma atualização residual é gerada, com todas as informações estáticas computadas por *Reduce*. Observe que o processamento das atualizações de um bloco de regras anotadas pode ser intercalado com o processamento de outros blocos ou mesmo regras condicionais anotadas. Cada elemento de `GSET` coleta um conjunto específico de atualizações estáticas, que serão disparadas quando o bloco associado se torna vazio.

Considerando novamente a afirmação 3 deste teorema, podemos ver que as funções externas eventualmente utilizadas nas atualizações dinâmicas de P^{an} serão chamadas com os mesmos valores tanto em P, no passo k, quanto em P_{in_1}, no passo $k+1$ Novamente, a justificativa para isso é o fato de que os valores das funções estáticas e dinâmicas possuem as mesmas interpretações nas duas especificações.

É fácil ver que os blocos associados aos elementos de `GSET` se tornam menores a cada passo, garantindo que se tornarão vazios em um número finito de passos. Quando isso acontece, as atualizações coletadas são disparadas em paralelo e registradas em `VFuncs`. É importante ressaltar que, devido ao pré-processamento descrito na Seção 5.2, cada bloco de regras anotadas contém **todas** as possíveis atualizações, estáticas e dinâmicas, de um determinado passo da execução P. Cada bloco contém as atualizações associadas a um determinado fluxo de controle, definido pela avaliação de uma seqüência de regras condicionais. Assim teremos, para toda atualização $f(t_1, \ldots, t_k)$ `:=` v pertencente a um bloco de regras anotadas, com $f \in \Upsilon^+$:

$$\texttt{VFuncs}(\langle f, \langle \mathit{Val}(t_1), \ldots, \mathit{Val}(t_k)\rangle\rangle) = \mathit{Val}(v)$$

Os passos a seguir constituem uma execução das regras da Figura 5.14. A função auxiliar `verif` é utilizada para determinar um possível valor `NewState` tal que

$$\texttt{VFuncs}(\langle f, t^* \rangle)) = \texttt{TSpec}(\langle f, t^* \rangle, \texttt{NewState}), \text{ para todo } f \in \Upsilon^+$$

Esse processo é conduzido pelo conjunto de regras executadas quando os valores 1, 2 e 3 são associados à função `BuildNewState`. Vamos assumir que esse trecho está corretamente codificado. Se não existe um valor que satisfaça a equação acima, então `NewState` será associado ao valor `TotS`+1. Quando `BuildNewState` é 4, um código residual da forma

$$\texttt{curstate := } x$$

é gerado, onde x é o valor corrente de `NewState`. Essa será a última regra residual associada ao bloco corrente.

Paralelamente à geração de código residual, duas outras ações são executadas. Se `NewState` tem o mesmo valor que `TotS`+1, isso significa que nenhuma posição da tabela `TSpec` contém exatamente os valores das funções estáticas armazendos em `VFuncs`. Se isso acontece, uma nova posição é criada em `TSpec`, armazenando os valores de `VFuncs`. Assim, para todo $l \in \texttt{Dom}(\texttt{1}, \texttt{VFuncs})$:

$$\texttt{TSpec}(l, \texttt{TotS} + 1) = \texttt{VFuncs}(l)$$

Observe que seria fácil incrementar o invariante deste teorema para mostrar que nunca duas posições da tabela `TSpec` conterão exatamente os mesmos valores. A outra ação que é executada consiste em reestabelecer os valores de `VFuncs`, antes do disparo das atualizações estáticas coletadas no bloco que foi processado. Isso é importante porque podem haver ainda outros blocos ou mesmo regras condicionais anotadas em `GSET` que ainda não foram processados.

O valor de `BuildNewState` se torna novamente 0, e o processamento das regras anotadas continua, até que `GSET` se torne vazio. Vamos analisar então os resultados nesse momento.

A especificação anotada P^{an} pode ser vista como uma árvore binária, onde os nodos internos são regras condicionais e as folhas são blocos de atualizações. Nas discussões acima, vimos que, durante a execução de *Spec* nos passos que constroem a regra residual executada no passo $k+1$ de P_{in_1}, a avaliação das regras condicionais estáticas resultam no mesmo fluxo de controle que o observado em P, no passo k. As regras condicionais dinâmicas, quando executadas no passo $k+1$ de P_{in_1}, também seguirão o mesmo fluxo de controle que o observado em P, no passo k. Isso faz com que as atualizações executadas no passo $k+1$ de P_{in_1} sejam as mesmas que as atualizações dinâmicas executadas no passo k de P. Assim, no passo seguinte de ambas as especficações, teremos

$$\mathcal{A}^{-}_{k+1} = \mathcal{A}^{res}_{k+2} - \{\texttt{curstate}^{\mathcal{A}^{res}_{k+2}}\}$$

Vimos que as regras da Figura 5.14 asseguram que existirá uma posição x em `TSpec` que contenha exatamente os valores das funções armazenadas em `VFuncs`, após o disparo das atualizações estáticas coletadas. Esses valores são equivalentes aos valores das funções estáticas de P^{an} no passo $k+1$. Essa posição x poderá ser uma já existente em `TSpec`, ou

foi criada no momento do processamento do bloco. De qualquer forma, esse valor inteiro será usado também na geração do código residual

$$\texttt{curstate} := x$$

Assim, a seguinte propriedade será válida:

$$f^{\mathcal{A}_{k+1}}(t^*) = \texttt{TSpec}(\langle f, t^* \rangle, \texttt{curstate}^{\mathcal{A}^{res}_{k+2}}), \text{ para todo } f \in \Upsilon^+$$

É fácil ver que os valores de `TSpec`, uma vez determinados, não são alterados nos passos seguintes da execução de *Spec*,.Logo podemos afirmar que o Teorema 10 é válido também para o estados $k+1$ da execução de P e $k+2$ da execução de P_{in_1}. ■

Com a demonstração desse teorema, mostramos que o algoritmo de especialização produz sempre especificações residuais corretas, se sua execução terminar em um número finito de passos.

6.3.4 Terminação do Algoritmo de Especialização

Na demonstração do Teorema 10, fizemos algumas afirmações sobre a terminação de partes do algoritmo de especialização da Seção 5.4, sem prova formal. Afirmamos que o conjunto `GSET` se torna vazio em um número finito de passos, após sofrer a inserção de um elemento associado à regra inicial ou à regra de transição da especificação anotada P^{an}. Esse afirmação é verdadeira e pode ser facilmente provada com técnicas semelhantes às utilizadas na Seção 6.2.1.

Mas outro possível ponto do algoritmo que pode gerar não terminação ocorre nas regras do lado direito da Figura 5.10. Nada garante que `CurS` será um valor inteiro maior ou igual a `TotS` em um número finito de passos. Quando uma regra condicional dinâmica é processada, dois elementos são inseridos em `GSET`, no lugar do elemento retirado. Até que `GSET` se torne vazio, várias novas entradas podem ter sido acrescentadas à tabela `TSpec`, e o valor de `TotS` poderá ser indefinidamente crescente.

Suponha que uma especificação P, com duas entradas, não termine em um número finito de pasos quando aplicada a um valor in_1 para a primeira entrada, independente do valor da segunda entrada. Então é razoável que o algoritmo de especialização também não termine, quando executado usando in_1 como entrada estática.

Mas uma situação inusitada pode ocorrer. Suponha que a especificação original P termine com um número finito de passos, se aplicada a entradas in_1 e in_2. Da forma como especificamos os algoritmos BTA e *Spec*, o algoritmo de especialização pode não terminar se aplicado à versão anotada P^{an} (considerando in_1 estática) e à entrada parcial in_1.

Uma análise de tempo de definição ideal é aquela que classifica o maior número de funções do programa original como estáticas, mas não adiciona possíveis casos de não terminação ao algoritmo de especialização. Esse é um problema não computável [55]. A maioria das abordagens que buscam garantir terminação de avaliadores parciais *offline* utiliza o conceito de valores intrinsicamente crescentes ou decrescentes [49, 53, 79]. O algoritmo de BTA apresentado na Seção 5.3 deveria ser melhorado usando técnicas semelhantes a essas, para garantir a terminação da especialização em um número finito de passos.

6.4 Conclusões

Neste capítulo, provamos algumas propriedades interessantes sobre os algoritmos apresentados no Capítulo 5. As demonstrações desenvolvidas sugerem que o modelo ASM, além de adequado para formalização de algoritmos, facilita a construção de demonstrações, principalmente usando indução matemática sobre os estados de uma execução. Algumas funções apresentadas no Capítulo 5 foram escritas usando-se o modelo funcional com avaliação estrita. Nesse caso usamos também a indução matemática para demonstrar propriedades sobre seu funcionamento.

As provas desenvolvidas estão relacionadas com:

1. pré-processamento da especificação ASM que serve de entrada para o avaliador parcial;
2. algoritmo de BTA;
3. algoritmo de especialização.

Nos dois primeiros casos, procuramos utilizar o modelo ASM de uma maneira mais formal. Na Seção 6.1, mostramos que o pré-processamento sobre a especificação de entrada ASM é um algoritmo que sempre termina em um número finito de passos, e produz uma nova especificação com a mesma semântica da original. Na Seção 6.2, mostramos que o algoritmo de BTA também sempre termina, produzindo uma divisão congruente que é utilizada em seguida para gerar uma especificação anotada.

Nas demonstrações relativas ao algoritmo de especialização, na Seção 6.3, enunciamos um teorema que associa a execução da especificação de entrada com a execução da especificação residual. O algoritmo de especialização *Spec*, executado sobre uma versão anotada da especificação, participa do teorema pois define o formato da especificação residual. A demonstração, desenvolvida de maneira mais informal quando relacionada à execução de *Spec*, prova que *Spec* produz especificações residuais corretas.

Algumas propriedades foram utilizadas, sem prova, na demonstração da correção de *Spec*. Essas propriedades podem ser demonstradas sem dificuldade, usando técnicas semelhantes às aplicadas nas Seções 6.1 e 6.2.

Um ponto que pode ser melhorado, como discutimos na seção anterior, é o algoritmo de BTA. As demonstrações nos mostraram que as técnicas simples aplicadas não garantem a terminação do algoritmo de especialização em um número finito de passos.

Capítulo 7

Extensões de Geração para ASM

Uma extensão de geração para um programa P é um programa que, se aplicado a parte da entrada de P, produz uma versão especializada de P. Um gerador de extensões de geração pode ser escrito à mão ou ser produzido automaticamente por meio da auto-aplicação de um avaliador parcial.

Neste capítulo, vamos apresentar um gerador de extensões de geração para a linguagem das Máquinas de Estado Abstratas, escrito à mão. Esse programa, que chamaremos `cogen` para ASM, processa especificações ASM contendo apenas regras básicas: atualizações, condicionais e blocos. O gerador de extensões de geração consiste de duas partes bem separadas. A primeira parte é um núcleo de rotinas que são utilizadas para a construção de uma extensão de geração a partir de uma representação abstrata de uma especificação ASM de entrada. A segunda parte são filtros, ou conversores, que permitem utilizar as rotinas do núcleo com uma linguagem ASM concreta específica.

Na Seção 7.1, discutimos a abordagem de extensão de geração, sem considerar especificamente o modelo ASM. Os detalhes da construção de `cogen` para ASM são apresentados na Seção 7.3.

7.1 Extensões de Geração

Na avaliação parcial tradicional, um programa e parte de sua entrada são submetidos a um avaliador parcial, que produz como saída um programa residual. Duas abordagens diferentes podem ser utilizadas: avaliação parcial *online* e *offline*. Os métodos diferem em relação ao momento em que os componentes do programa são classificados como estáticos e dinâmicos: junto com a execução da especialização, na abordagem online, e em uma fase anterior, denominada BTA (análise de tempo de definição), na abordagem offline.

Uma técnica diferente se tornou popular na década de 90, denominada abordagem de *extensões de geração*. 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 como saída um *extensão de geração* para P.

A Figura 7.1 exibe um esquema de uso de `cogen`. Comparando com a Figura 3.7, do Capítulo 3, podemos ver que a abordagem de extensões de geração é semelhante à

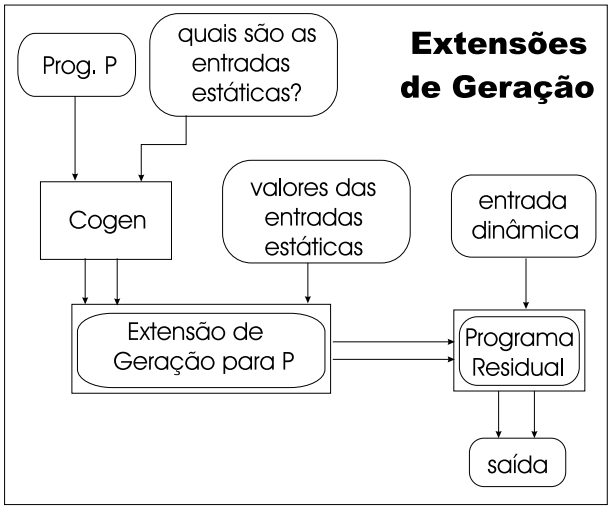


Figura 7.1: Esquema de Uso de um *Gerador de Extensões de Geração*.

abordagem offline de avaliação parcial tradicional, na medida que um novo programa é gerado usando apenas as informações sobre quais entradas são estáticas. No método offline, um programa anotado é gerado pela fase de BTA, o qual é submetido em seguida ao especializador, juntamente com os valores das entradas estáticas. Por outro lado, `cogen` gera um programa que pode ser executado sobre as entradas estáticas, sem a necessidade de um especializador. Desse modo, `cogen` pode ser visto como um especializador, especializado em relação a um programa e parte de sua entrada.

7.1.1 Definição Equacional

No Capítulo 3, usamos a seguinte definição equacional para o comportamento de um avaliador parcial `mix`:

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

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.

As equações acima são mais adequadas para definir o funcionamento de um avaliador parcial online. No método offline, o processamento é geralmente dividido nas fases `BTA`

(análise de tempo de definição) e `spec` (especialização):

$$P^{an} = [\![\texttt{BTA}]\!]_L\,(P,\ \{in_1 \approx static,\ in_2 \approx dynamic\})$$
$$P_{in_1} = [\![\texttt{spec}]\!]_L\,(P^{an}, in_1)$$

A fase de BTA gera um programa anotado P^{an}. As anotações são seguidas pelo especializador para produzir o programa residual. A entrada para `BTA` é o programa P e informações que indiquem quais entradas são estáticas, fornecidas pelo usuário.

Um gerador de extensões de geração `cogen` é executado sobre as mesmas entradas que o programa `BTA`. Mas a extensão de geração P-*gen* produzida pode ser executada diretamente sobre as entradas estáticas:

$$P\text{-}gen = [\![\texttt{cogen}]\!]_L\,(P,\ \{in_1 \approx S,\ in_2 \approx D\})$$
$$P_{in_1} = [\![P\text{-}gen]\!]_S\,(in_1)$$

Observe como as equações sugerem que P-*gen* seja uma versão especializada de `spec`.

7.1.2 Avaliação Parcial Versus Extensões de Geração

As abordagens de avaliação parcial tradicional e extensões de geração são, sob um certo ponto de vista, equivalentes. Para mostrar isso, vamos recordar as três projeções de Futamura:

$$\begin{aligned}[\![\texttt{mix}]\!]_L\,(P, in_1) &= P_{in_1}\\ [\![\texttt{mix}]\!]_L\,(\texttt{mix}, P) &= P\text{-}gen\\ [\![\texttt{mix}]\!]_L\,(\texttt{mix}, \texttt{mix}) &= \texttt{cogen}\end{aligned}$$

Se P é um interpretador de uma linguagem *Lint* e in_1 é um programa escrito nessa linguagem, então P_{in_1} é o resultado da compilação de in_1. A linguagem objeto será T, a linguagem dos programas gerados pelo avaliador parcial. O programa P-*gen* é um compilador de *Lint* para T e `cogen` é 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 tradicional. Por outro lado, `cogen` pode ser usado para obter os mesmos resultados da avaliação parcial tradicional, com um passo adicional, como visto na Seção 7.1.1. Então qual das abordagens 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. Como vimos na Seção 3.5.2, algumas razões justificam 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 estaticamente tipadas. Vamos nos concentrar nesse problema a seguir. Nem o gerador de extensões de geração, nem as extensões de geração produzidas, precisam conter um meta-interpretador.

Na Seção 3.5.2, discutimos um problema que ocorre quando se escreve um interpretador para uma linguagem estaticamente tipada. Um único tipo universal deve ser utilizado no interpretador para representar um número ilimitado de tipos utilizado pelos programas que são interpretados. O mesmo é válido para um avaliador parcial auto-aplicável: um único tipo universal deve ser utilizado para representar a entrada estática do programa que está sendo especializado. Isso pode trazer sérios problemas de ineficiência, como foi apresentado na Seção 3.5.2. Vamos analisar aqui esse problema de uma maneira mais formal.

Vamos supor que `mix` seja um avaliador parcial para uma linguagem estaticamente tipada. O programa P que `mix` vai especializar pode ter qualquer tipo de entrada, por exemplo inteiros, caracteres etc. Assim, a entrada estática deve ser codificada em um único tipo universal *Value*. O próprio programa P deve ser codificado em um tipo que iremos chamar de *Pgm*. Vamos usar a notação $\overline{d}^{t}$ para denotar uma representação de um dado d (de um tipo qualquer) dentro de uma estrutura de dados de tipo t. Levando em conta essa codificação, as projeções de Futamura podem ser reescritas da seguinte forma:

$$\begin{aligned}
[\![\texttt{mix}]\!]_L(\overline{P}^{Pgm}, \overline{in_1}^{Value}) &= \overline{P_{in_1}}^{Pgm} \\
[\![\texttt{mix}]\!]_L(\overline{\texttt{mix}}^{Pgm}, \overline{\overline{P}^{Pgm}}^{Value}) &= \overline{P\text{-}gen}^{Pgm} \\
[\![\texttt{mix}]\!]_L(\overline{\texttt{mix}}^{Pgm}, \overline{\overline{\texttt{mix}}^{Pgm}}^{Value}) &= \overline{\texttt{cogen}}^{Pgm}
\end{aligned}$$

Observe que o programa P é duplamente codificado na segunda projeção. O mesmo acontece com o próprio `mix` na terceira projeção, quando `cogen` é gerado automaticamente. Isso pode resultar em uma estrutura de dados muito grande que traz ineficiência para a auto-aplicação.

Um gerador de extensões de geração escrito à mão, por outro lado, tem como entrada e saída um programa, não precisando de lidar com os tipos de dados referenciados no programa de entrada. As declarações de tipo do programa de entrada são simplesmente copiadas na extensão de geração produzida.

7.1.3 Dificuldades com Auto-Aplicação de `mix`

Na Seção 5.5, discutimos aspectos envolvendo a auto-aplicação de um avaliador parcial para ASM, que designamos $\texttt{mix}_{ASM}$. Vimos que, para conseguirmos bons resultados com a especialização de $\texttt{mix}_{ASM}$, foi necessário utilizar comandos *var* na sua codificação. Mas não fomos capazes de introduzir em $\texttt{mix}_{ASM}$ um processamento eficiente desses comandos *var*. O avaliador parcial processa apenas comandos ASM básicos.

O problema descrito acima torna impossível que $\texttt{mix}_{ASM}$ processe seu próprio texto. Uma solução para gerar compiladores, usando a Segunda Projeção de Futamura, foi submeter $\texttt{mix}_{ASM}$ a um avaliador parcial mais poderoso, o qual processa inclusive comandos *var*.

Entretanto, a geração de um gerador de compiladores `cogen`, usando a Terceira Projeção de Futamura, tornou-se inviável.

A abordagem de extensões de geração consiste em escrever `cogen` à mão, assim os problemas de auto-aplicação são contornados. Como vimos anteriormente, `cogen` pode ser usado para produzir extensões de geração, compiladores a partir de interpretadores e também pode ser usado para avaliação parcial tradicional. Na Seção 7.3, apresentaremos os detalhes da construção de um gerador de extensões de geração para ASM.

7.2 A Linguagem Xasm

As experiências usando extensões de geração e ASM que conduzimos usaram como linguagem concreta a linguagem Xasm. Essa linguagem foi criada por Matthias Anlauff [6].

Além de comandos básicos ASM, a linguagem Xasm 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.

7.2.1 Características Importantes

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. Isso é uma característica fundamental para gerenciamento de aplicações volumosas.

A linguagem permite a execução de comandos seqüencialmente, por meio de um construtor especial. Isso viola o princípio básico das ASM, que define a execução de comandos em paralelo, tornando inválidas as técnicas de demonstração de propriedades geralmente utilizadas no modelo. Entretanto, é bastante útil para simplificar o texto de especificações geradas automaticamente, como é o caso das extensões de geração.

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 uma das aplicações mais importantes do modelo ASM, que é a descrição da semântica de linguagens de programação.

7.2.2 Xasm e Cogen

Como será explicado na Seção 7.3, o gerador de extensões de geração `cogen` para ASM é dividido em duas partes. Uma parte são as rotinas do núcleo, que funcionam independentemente da linguagem concreta utilizada. Outra parte são os filtros que permitem o acoplamento do núcleo a uma linguagem ASM concreta específica.

Construímos um filtro que processa um subconjunto da linguagem Xasm. Nas especificações de entrada, são aceitos apenas comandos ASM básicos para serem especializados.

Abstrações de funções são copiadas diretamente para a extensão de geração, assim podem conter quaisquer tipos de comandos. A descrição de gramáticas são também copiadas diretamente para a extensão de geração. Livres das restrições impostas às especificações de entrada, utilizamos ao máximo as facilidades oferecidas pela linguagem nas extensões de geração produzidas, inclusive a execução de comandos seqüencialmente. Vamos descrever alguns aspectos do funcionamento do filtro quando discutirmos o funcionamento das rotinas do núcleo de `cogen`.

7.2.3 Um Interpretador escrito em Xasm

A Figura 7.2 exibe um interpretador para uma versão da Máquina de Turing, escrito em Xasm. Esse interpretador é muito semelhante ao apresentado no Exemplo 5 da Seção 2.2. Vamos utilizar esse exempo nas próximas seções.

A primeira linha da especificação exibe o nome da mesma e os parâmetros. Nesse caso, os parâmetros são nomes de arquivos que armazenam um programa MT (`progfile`) e o estado inicial da fita (`tapefile`). Três funções externas agem sobre os parâmetros: `TMparse`, `READTAPE` e `PRINTTAPE`. Seu código não é exibido na Figura 7.2.

Observe a regra de inicialização, inserida em uma seção denominada `init`. A função `READTAPE` lê o estado inicial da fita, que deve estar descrito no arquivo de nome `tapefile`. Esse valor inicial é armazenado na função `tape`, que permite acesso a cada caractere da fita. Retorna o valor inicial da cabeça de leitura/gravação, que será armazenado em `thead`.

A função `TMparse` está associada à gramática declarada na especificação. Lê o arquivo cujo nome é fornecido e executa a análise sintática e as ações associadas às produções da gramática. A gramática também não é exibida na Figura 7.2, mas suas ações definem valores para as funções `prog_instr`, `prog_par1` e `prog_par2` que representam o código e os parâmetros de cada instrução MT. A função `pc` indica a instrução corrente.

Após a regra de inicialização, pode-se ver a regra de transição da especificação. Ações adequadas estão associadas a cada instrução MT. A instrução `STOP` indica o fim da interpretação do programa MT, com a exibição do estado final da fita.

7.3 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 $\mathcal{S}$.

***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

```
asm TURING (progfile, tapefile)

external function TMparse (String) -> prog
external function READTAPE (String) -> Int
external function PRINTTAPE -> Bool
universe Instr = {LEFT, RIGHT, WRITE, GOTO, IFGOTO, STOP}
function prog_instr (Int) -> Instr
function prog_par1 (Int) -> Char
function prog_par2 (Int) -> Int
function tape (Int) -> Char
function pc -> Int
function thead -> Int
//... declaração de outras 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
  auxb := PRINTTAPE
  exit := 1
endif

endasm
```

Figura 7.2: Interpretador para MT escrito em Xasm.

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.

O primeiro passo executado pelo núcleo de `cogen` é um pré-processamento da especificação de entrada $\mathcal{S}$, seguido da análise de tempo de definição. Após a BTA, uma nova reorganização das regras é conduzida, e finalmente uma representação abstrata da extensão de geração é produzida. Os algoritmos serão descritos usando o próprio modelo ASM e o paradigma funcional com avaliação estrita. Para isso precisamos então de estabelecer uma notação para a representação da especificação de entrada e da extensão de geração que será produzida. Essa notação é discutida a seguir, na Seção 7.3.1. As seções seguintes descrevem o comportamento do núcleo de `cogen`.

A descrição do comportamento do núcleo de `cogen` usando o próprio modelo ASM, apresentada nas Seções 7.3.2 e 7.3.3, não sugere qualquer intenção de auto-aplicação. Ao contrário do que acontece no Capítulo 5, o único objetivo é deixar claro o funcionamento dos algoritmos.

7.3.1 Representação de Especificações

As especificações de entrada são representadas da mesma forma que a utilizada na Seção 5.1, por meio dos domínios *FNAME*, *TERM* e *RULE* e os construtores *term*, *update*, *block* e *cond*. Apenas mais um construtor será adicionado para que o domínio *RULE* possa ser utilizado também na representação abstrata da extensão de geração, como veremos a seguir.

A representação das regras da extensão de geração utiliza um novo domínio, chamado *GERULE*, e os construtores *condsta* e *dsnode*:

$$\begin{aligned} condsta &: (TERM \times GERULE \times GERULE) \rightarrow GERULE \\ dsnode &: RULE^* \times RULE \rightarrow GERULE \\ gotostate &: INT \rightarrow RULE \end{aligned}$$

Na Seção 7.3.3, veremos exatamente como esses construtores serão utilizados. Como discutimos acima, mais um novo construtor *gotostate* também será introduzido, para que o domínio *RULE* possa ser utilizado na representação abstrata da extensão de geração.

A extensão de geração é uma especificação que gera como saída outra especificação ASM. Assim, dentro de seu código existem chamadas para funções 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.

Não vamos apresentar aqui todas as funções externas usadas na geração de especificações residuais, mas vamos discutir seu funcionamento geral. Todas essas funções retornam

números inteiros, que são usados como identificadores únicos na interface com a extensão de geração. Por exemplo, qualquer função referenciada no programa residual deve ser antes declarada usando

$$GenDECL : STRING \times STRING \rightarrow INT$$

onde o primeiro parâmetro é o nome da função e o segundo é o texto de sua declaração original, na especificação de entrada $\mathcal{S}$. O valor inteiro retornado é utilizado em todos os pontos seguintes em que a função é referenciada. Listas são representadas por meio de elos construídos pelos identificadores inteiros, terminadas por uma chamada a *GenNIL*. São utilizadas na representação de parâmetros em uma chamada de função e em blocos de regras. Assim, para produzir uma chamada de função `f(x)` no programa residual, a extensão de geração deverá conter um código similar a:

```
__f := GenDECL (''f'', ''function f ...'')
__x := GenDECL (''x'', ''function x ...'')
...
...  GenFUNC (__f, GenPARS (GenFUNC (__x, GenNIL), GenNIL)) ...
```

As funções externas *GenFunc* e *GenPars* produzem, respectivamente, uma chamada de função e uma lista de parâmetros. Observe que `__f` e `__x` armazenam números inteiros que servem como identificadores únicos para as funções residuais de nome `f` e `x`. Veremos mais exemplos do uso dessas funções externas nas seções seguintes.

7.3.2 Pré-Processamento e BTA

Uma boa parte do processamento inicial do núcleo de `cogen` se assemelha ao comportamento de um avaliador parcial tradicional. Assim, algumas das rotinas serão idênticas às descritas no Capítulo 5.

Inicialmente, a especificação de entrada $\mathcal{S}$ é pré-processada da mesma forma que a descrita na Seção 5.2. Esse pré-processamento, como visto anteriormente, não afeta a regra de inicialização, mas apenas a regra de transição da especificação. A regra de transição é transformada em uma árvore cujos nodos interiores são regras condicionais e as folhas são blocos de atualizações.

Em seguida, uma análise de tempo de definição é executada, tendo como entradas a especificação pré-processada e uma definição de quais funções de entrada de $\mathcal{S}$ são estáticas e dinâmicas. A BTA produz uma divisão das funções referenciadas mas agora não gera uma especificação anotada. O processo utilizado é a interpretação abstrata, o mesmo descrito na Seção 5.3. O resultado da divisão é armazenado em uma função `bta_val`.

Após a 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. Supondo que a regra de transição tem o formato de uma árvore, o processamento faz com que todos as regras condicionais estáticas sejam visitadas antes das dinâmicas, em um caminhamento da raiz para as folhas. A função `downdyn`, exibida na Figura 7.3, descreve esse processamento. A representação da especificação de entrada utiliza as mesmas estruturas que foram sugeridas na Seção 5.1.

```
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) = BTANEG 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) = BTAPOS 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) = BTAPOS 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
```

Figura 7.3: Função `downdyn`.

7.3.3 Representação Abstrata da Extensão de Geração

A especificação modificada pelo processamento discutido na Seção 7.3.2 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 7.3.1.

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 exatamente para a extensão de geração, assim não farão parte da especificação residual. São representados por instâncias do construtor *condsta*. Observe que 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

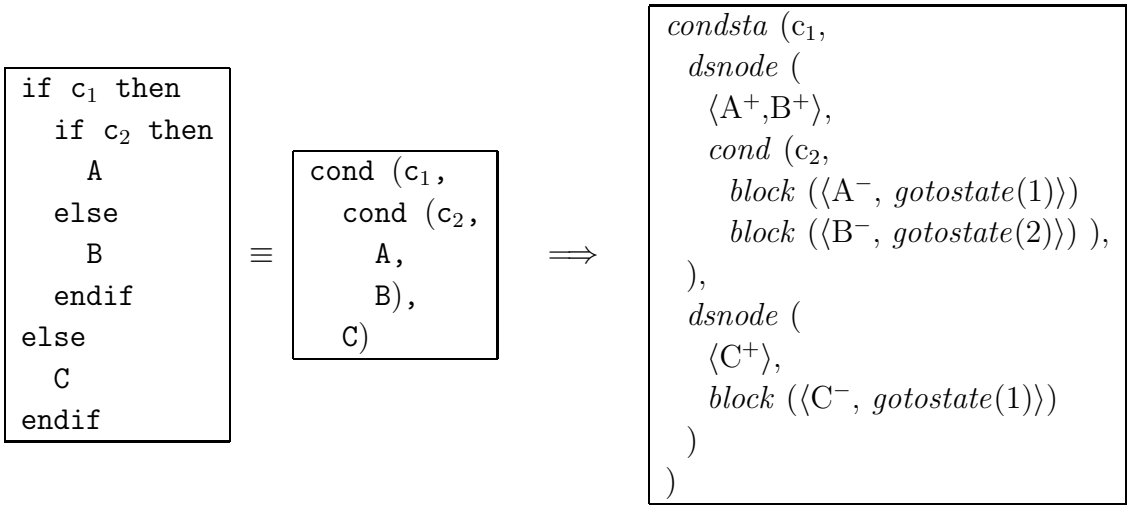


Figura 7.4: Representação Abstrata em uma Extensão de Geração.

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 uma 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 7.4, 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. Esse é o resultado do processamento descrito nesta seção. Para indicar separadamente as atualizações positivas e negativas 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 positivas de $\mathtt{A}^+$ e $\mathtt{B}^+$. O código residual gerado é uma modificação da regra condicional

```
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) = BTAPOS 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) = BTAPOS then
    condsta (term(f,t), sds(0, r1), sds(0, 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
```

Figura 7.5: Função sds.

que contém c_2. Observe que elementos *gotostate* são inseridos, com um valor inteiro que indica a posição na lista $\langle A^+, B^+ \rangle$ a que estão associados.

O algoritmo que descreve o processamento discutido nesta seção é definido pela função sds, exibida na Figura 7.5. A função sds, além de uma regra ASM, tem um argumento adicional que é um valor inteiro. A chamada inicial a sds pode ser feita com qualquer valor para esse argumento, que é utilizado para determinar a correta associação de elementos *gotostate* a seus blocos relacionados.

7.3.4 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 7.3.3. 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++, usando o paradigma de orientação a objetos. Uma regra ASM é definida como uma classe que tem subclasses representando as regras de atualização, condicionais e blocos. 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*.

A construção da extensão de geração depende fortemente da linguagem concreta sendo usada, mas podemos citar alguns pontos que deverão ser seguidos na maioria dos filtros. A seguir, procuramos enumerar esses pontos na ordem provável em que aparecem no texto do gerador de extensões. Vamos chamar a especificação de entrada de $\mathcal{S}$ e a extensão de geração produzida de $\mathcal{G}$:

1. No cabeçalho de $\mathcal{S}$, os parâmetros de entrada são especificados de alguma forma. O cabeçalho de $\mathcal{G}$ deve conter apenas os parâmetros estáticos.

2. As construções que não estão relacionadas diretamente às regras ASM, como definição de gramáticas em Xasm, devem ser copiadas para o texto de $\mathcal{G}$.

3. Se a linguagem concreta necessita de declarações de funções, as declarações das funções estáticas são copiadas para $\mathcal{G}$. Nesse caso, devem ser também produzidas declarações para:

   (a) Funções externas que geram código residual.

   (b) Representação das funções dinâmicas usando valores inteiros (ver Seção 7.3.1).

   (c) Funções usadas pela própria extensão de geração para implementar o algoritmo de especialização polivariante. Isso inclui, de alguma forma, as funções estáticas de $\mathcal{S}$.

4. A regra de inicialização de $\mathcal{G}$ deve executar os seguintes procedimentos:

(a) Gerar código para a construção do cabeçalho do programa residual, que contém apenas os parâmetros dinâmicos de $\mathcal{S}$.

(b) Gerar código residual correspondente às declarações das funções dinâmicas de $\mathcal{S}$.

(c) Gerar código residual correspondente aos componentes dinâmicos da regra de inicialização de $\mathcal{S}$.

(d) Inicializar suas próprias funções, que incluem as funções estáticas de $\mathcal{S}$.

5. A regra de transição de $\mathcal{G}$ é gerada seguindo as diretrizes apresentadas na Seção 7.3.3. As regras condicionais estáticas são equivalentes às utilizadas em $\mathcal{S}$. Quando um elemento *dsnode* é encontrado, é produzido código para:

   (a) Disparar, seqüencialmente, cada conjunto de atualizações estáticas definidas no primeiro componente do elemento *dsnode*.

   (b) Verificar se cada novo estado já está armazenado na tabela de estados. Se não, armazená-lo.

   (c) Gerar código residual correspondente à regra dinâmica, definida pelo segundo componente do elemento *dsnode*. Isso inclui gerar código apropriado para os elementos *gotostate*.

6. Finalmente, quando a execução de $\mathcal{G}$ termina, uma representação abstrata de uma especificação residual foi inteiramente gerada. O núcleo oferece rotinas para realizar uma transição de compressões sobre essa especificação residual. A extensão de geração $\mathcal{G}$ pode executar uma chamada a essas rotinas e em seguida traduzir a representação abstrata para uma representação textual, escrita na linguagem concreta utilizada.

A Seção 7.4 a seguir apresenta um exemplo de utilização do filtro para a linguagem Xasm que desenvolvemos.

7.4 Experimentos com Cogen e Xasm

Como discutido anteriormente, construímos um filtro que permite utilizar a linguagem Xasm com as rotinas do núcleo de `cogen`. Todas os experimentos que conduzimos utilizaram esse filtro.

O nosso primeiro experimento envolveu um interpretador para uma linguagem muito simples, uma versão da máquina de Turing. Em seguida, construímos diversos interpretadores para linguagens imperativas simples, até que finalmente construímos um interpretador para um significativo subconjunto da linguagem C. Um compilador de C para Xasm foi automaticamente gerado e testado. Uma parte desses experimentos é descrita nas seções seguintes.

```
asm GEN-TURING (progfile)

function __tapefile -> Int
function __READTAPE -> Int
function __PRINTTAPE -> 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
```

Figura 7.6: Declarações de Funções no Compilador de MT para Xasm.

7.4.1 Compilador para Máquina de Turing

O interpretador para uma versão da linguagem da máquina de Turing, cujo código é parcialmente exibido na Figura 7.2, foi submetido ao gerador 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 uma extensão de geração que é um compilador da linguagem MT para Xasm. O cabeçalho e parte das declarações do compilador são exibidos na Figura 7.6. 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, da maneira como foi descrito na Seção 7.3.1. 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. Haveriam tantas funções como `__TAB_pc` quantas fossem as funções que fizessem parte dos estados estáticos produzidos pela extensão de geração.

Um pequeno trecho de código da regra de transição do compilador é exibido na Figura 7.7. 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 va-

```
...
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)) )
```

Figura 7.7: Trecho de Código do Compilador de MT para Xasm.

lor 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`.
2. Se não for encontrado, uma nova instância é adicionada a `__TAB_pc`, armazenando o valor de `pc`.
3. Em qualquer dos casos, a função `newstate(`i`)` conterá a posição da tabela com o valor de `pc`.
4. 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 nas Seções 7.3.3 e 7.3.4. 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.

A especificação residual, de maneira similar à descrita no Capítulo 5, 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 $k \in$ `Dom(GenRESRULE)`. O fluxo de controle é determinado por atualizações sobre `curstate`, as quais são geradas pelas chamadas a `GenGOTO`.

7.4.2 Compilador para um Subconjunto da Linguagem C

Para realizarmos experimentos mais práticos que o apresentado na Seção 7.4.1, escrevemos, em Xasm, um interpretador para um significativo subconjunto da linguagem C.

Esse interpretador utiliza idéias semelhantes às apresentadas em [42]. É possível submeter à interpretação programas em C com:

- expressões com operadores aritméticos básicos, auto-incremento, auto-decremento e chamada de funções;
- comando de seleção IF e comando de repetição WHILE;
- declarações de variáveis em blocos;
- declaração e chamada recursiva de procedimentos e funções com parâmetros;
- declarações e uso de registros;
- uso de apontadores, endereços e aritmética de apontadores.
- alocação de memória dinâmica.

Como em [42], 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 ações que são executadas, juntamente com a definição do novo valor de `CurTask`.

Para utilizar a representação descrita acima, o programa fonte em C é traduzido para uma forma intermediária. O interpretador escrito em Xasm contém uma seção que descreve uma gramática do subconjunto de C considerado. Essa gramática permite a geração automática de um analisador léxico e sintático para os programas fontes, e as ações associadas às regras realizam a tradução desses programas para a forma intermediária. Podemos dizer que a gramática e as ações semânticas associadas implementam parte da semântica estática da linguagem, enquanto que as regras ASM do interpretador implementam sua semântica dinâmica.

BTA e Chamadas Recursivas de Funções

Um problema muito comum na definição de interpretadores para linguagens com chamadas recursivas de funções teve que ser enfrentado, envolvendo a classificação da função `CurTask` na fase de BTA. 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. Na análise de tempo de definição, como sempre, é desejável classificar o maior número de funções como estáticas. Para obter bons resultados na especialização, é imprescindível ter a função `CurTask` classificada como estática, mas isso às vezes requer a aplicação de alguns truques, como descrito a seguir.

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, em uma chamada de uma função f, o valor de `CurTask`

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

Tabela 7.1: Testes com o Interpretador de C.

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%

Tabela 7.2: Testes com o Compilador de C para Xasm.

também 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 algortimo apresentado na Seção 7.3.2, CurTask deveria ser classificada como dinâmica, o que iria produzir resultados muito fracos na especialização.

Existe uma forma simples de contornar esse problema, que de tão utilizada passou a ser conhecida como "O Truque" [55]. Analisando o código do programa P, é fácil determinar os possíveis valores assumidos por `CurTask` no retorno de cada função. O número desses valores é finito e pode ser determinado usando apenas P, que é uma informação estática. Na tradução para a forma intermediária, nos pontos em que uma função f retorna, é inserida uma seqüência de instruções do tipo "`return` $\mathtt{r}_i$", uma instrução para cada possível ponto de retorno $\mathtt{r}_i$ de f. O interpretador de C interpreta essa construção da seguinte forma: "`if Stack(index) =` $\mathtt{r}_i$ `then CurTask :=` $\mathtt{r}_i$ `endif`", 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.

Esse truque é utilizado no interpretador de C desenvolvido, tornando-o mais adequado à avaliação parcial.

Testes

As Tabelas 7.1 e 7.2 mostram 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`. O interpretador

```
int fib (int x) {
  if (x <= 1)
    return 1;
  return fib(x-1) + fib(x-2);
}

void main() {
  PrintInt (fib(14));
}
```

Figura 7.8: Programa P_1.

foi executado sobre vários programas C. A Tabela 7.1 mostra medidas de tempo para a execução de dois desses programas. 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. O código desses programas é exibido nas Figuras 7.8 e 7.9. As funções `PrintInt` e `ReadInt` são usadas, respectivamente, para impressão e para leitura de um valor inteiro.

Em seguida, um compilador de C para Xasm foi gerado, usando o gerador de extensões de geração `cogen` desenvolvido. Chamamos esse compilador de `CtoXasm`. Os mesmos programas P_1 e P_2 foram então compilados para Xasm e depois um executável foi gerado usando `XasmC`. A Tabela 7.2 mostra medidas de tempo para essas tarefas e para a execução dos programas compilados.

Comparando o tempo de execução dos programas P_1 e P_2, quando compilados usando `CtoXasm`, com o tempo de interpretação dos mesmos programas pelo interpretador de C, observamos uma melhoria de 42%. Essa melhoria está aquém do que esperávamos. Uma das causas que contribuíram para isso é o fato de Xasm não ser estaticamente tipada, e a abordagem de extensões de geração é mais adequada a esse tipo de linguagem.

Os resultados ainda estão longe de poderem ser comparados a um compilador real. Por exemplo, usando uma mesma plataforma para testes, os programas P_1 e P_2, compilados com o compilador `gcc`, executaram aproximadamente 1000 vezes mais rápido do que quando compilados pelo nosso sistema. Convém lembrar que, para produzir uma versão executável para os programas P_1 e P_2, nosso sistema deve usar primeiro o compilador gerado por `cogen` e em seguida o compilador de Xasm para código executável. Assim, dependemos fortemente da eficiência do compilador oferecido pela linguagem Xasm.

7.5 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

```
void main() {
int n=0, A[100], x, i, j, menor, aux;
  ReadInt (&x);
  while (x >= 0) {
    A[n++] = x; ReadInt (&x);
  }
  i = 0;
  while (i < n) {
    menor = i; j = i+1;
    while (j < n) {
      if (A[j] < A[menor]) menor = j;
      j++;
    }
    if (menor != i) {
      aux = A[i]; A[i] = A[menor]; A[menor] = aux;
    }
    i++;
  }
  i = 0;
  while (i < n)
    PrintInt (A[i++]);
}
```

Figura 7.9: Programa P_2.

originais, mais genéricos. Uma área onde essas técnicas têm sido utilizadas com sucesso é na geração de compiladores dirigida por semântica [58, 36, 24, 60, 61, 48]. 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 alternativa mais indicada é escrever o gerador de compiladores à mão. Essa abordagem é conhecida como abordagem de extensões de geração.

Neste capítulo, apresentamos um gerador de extensões de geração `cogen` para a linguagem ASM. 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. O programa `cogen` é usado para produzir extensões de geração para especificações ASM. Se a especificação é um interpretador de uma linguagem L, então um compilador de L para ASM é automaticamente gerado.

Exibimos os algoritmos que implementam o núcleo do gerador de extensões de geração usando a própria linguagem 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 7.4.2 mostram melhoras no tempo de execução de especificações usando um compilador automaticamente gerado. Consideramos que o fato de Xasm não ser uma linguagem estaticamente 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, denominada Machĭna[81]. Como é uma linguagem estaticamente 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.

Capítulo 8

Conclusões

As Máquinas de Estado Abstratas são usadas com sucesso para a descrição da semântica de linguagens de programação [20, 21, 22, 42, 84], e para descrição do funcionamento de muitos outros sistemas. As especificações são simples de serem entendidas e podem ser apresentadas no nível de abstração desejado [15]. Entretanto, como a maioria dos modelos para descrição de semântica, ASM sofre com dificuldades relacionadas à eficiência na execução das especificações.

O principal objetivo do nosso trabalho de pesquisa é a utilização de técnicas de avaliação parcial para produzir especificações ASM mais eficientes. Avaliação parcial é uma técnica que contribui para a eficiência na execução de programas por meio da geração automática de versões especializadas desses programas. Quando usada com interpretadores, essa técnica permite a geração automática de programas compilados e de compiladores. A descrição da semântica de uma linguagem em ASM é geralmente dada por um interpretador para essa linguagem. Assim, avaliação parcial parecia ser, desde o início, uma abordagem promissora para a geração de especificações ASM mais eficientes e para a geração automática de compiladores dirigida por semântica.

Neste capítulo vamos rever alguns passos do desenvolvimento do nosso trabalho de pesquisa e discutir os resultados alcançados. Vamos também apresentar os nossos planos para a continuação do trabalho.

8.1 O Trabalho de Pesquisa Desenvolvido

As primeiras experiências envolvendo ASM e avaliação parcial foram conduzidas por Huggins e Gurevich [43, 51]. Um avaliador parcial para uma linguagem baseada no modelo ASM foi desenvolvido, escrito na linguagem C. Experiências muito simples com compilação dirigida por semântica são discutidas. Como a auto-aplicação não pôde ser explorada, geração automática de compiladores não é um assunto considerado nesse trabalho.

Nosso trabalho de pesquisa iniciou em 1997, com os primeiros contatos com o modelo ASM. Em 1998, participamos da Escola de Verão da Universidade de Copenhagen, cujo tema foi “Avaliação Parcial - Teoria e Prática”. O evento reuniu os maiores pesquisadores do assunto, que apresentaram tópicos introdutórios e avançados. Após esse evento, foram

desenvolvidos nossos primeiros experimentos com aplicação de técnicas de avaliação parcial ao modelo ASM.

Quando iniciamos o trabalho, o mais difundido sistema integrado para definição e execução de especificações ASM era o desenvolvido por Del Castillo [26]. Nesse sistema, a linguagem utilizada obedece fielmente o modelo ASM definido por Gurevich em [39, 40]. As especificações são interpretadas, e é possível visualizar os estados gerados passo a passo. Entretanto, esse sistema estava muito instável e incompleto. Particularmente, uma definição da linguagem utilizada não estava disponível.

Como nenhum sistema atendia às nossas necessidades, definimos uma linguagem simples que seguia o modelo ASM e construímos um interpretador para a mesma, escrito em Java. Nossas primeiras experiências com avaliação parcial utilizaram essa linguagem e esse interpretador. A linguagem adotada é não tipada, implementa as regras básicas do modelo (atualização, condicional e blocos) e algumas regras não básicas, como a regra *var*. Possibilita também a definição de funções usando o paradigma funcional com avaliação estrita.

Construímos um avaliador parcial `mix` para a linguagem adotada, escrito também em Java. Algumas técnicas de avaliação parcial específicas para o modelo ASM são aplicadas. Algumas são semelhantes às apresentadas em [43, 51], outras são completamente novas, devido à linguagem ser mais sofisticada que a utilizada por Huggins. Como a linguagem possui uma parte que obedece o paradigma funcional com avaliação estrita, o avaliador parcial utiliza também técnicas de avaliação parcial para esse paradigma. Para conduzir experimentos envolvendo geração de compiladores, por meio da segunda projeção de Futamura, desenvolvemos uma versão de `mix` escrita na própria linguagem ASM. As conclusões relativas a esses experimentos são apresentadas na Seção 8.2.

Enfrentamos alguns problemas relacionados à auto-aplicação do avaliador parcial, o que impediu a geração automática de um gerador de compiladores, por meio da terceira projeção de Futamura. Nesse mesmo período, participamos da definição de uma nova linguagem baseada no modelo ASM, designada *Machĭna* [81]. A linguagem Machĭna, que permite a definição de tipos estaticamente associados às funções ASM, foi projetada com o objetivo de facilitar uma compilação eficiente para linguagens imperativas comuns. A direção natural do trabalho seria então a adoção da *abordagem de extensões de geração*. Essa abordagem permite a geração de compiladores sem os problemas relacionados à auto-aplicação de avaliadores parciais, e é adequada a linguagens estaticamente tipadas [70].

Um compilador de Machĭna para C foi um projeto desenvolvido paralelamente ao nosso trabalho de pesquisa, por outros membros do grupo do *Laboratório de Linguagens do DCC-UFMG*. Após um estágio de quatro meses na Universidade de Copenhagen, onde estudamos a abordagem de extensões de geração e outros tópicos, estávamos prontos para concluir a implementação de um *gerador de extensões de geração* para ASM. Entretanto, ainda não estava disponível uma versão estável do compilador de Machĭna para C. Decidimos então implementar o gerador de extensões de geração em duas partes bem separadas:

1. um núcleo, que processa regras ASM básicas e funciona de modo independente da linguagem ASM concreta utilizada.

2. filtros que servem de interface entre o núcleo e uma linguagem ASM concreta.

Para nossos experimentos, construímos um filtro para a linguagem Xasm [6], que possui um compilador para C bastante utilizado. Na Seção 8.3, apresentamos as conclusões relacionadas aos experimentos com a abordagem de extensões de geração.

8.2 Avaliação Parcial e ASM

A linguagem adotada nos experimentos de avaliação parcial se baseia no modelo ASM, mas permite também a definição de funções usando o paradigma funcional com avaliação estrita. Assim, o avaliador parcial `mix` para ASM construído combina técnicas de avaliação parcial especialmente desenvolvidas para o modelo ASM com técnicas utilizadas em linguagens funcionais estritas.

Um teste foi utilizado para mostrar que `mix` possui um desempenho satisfatório, ou seja, o máximo de informação estática é computada durante a fase de especialização e não aparece nas especificações ASM residuais. Esse teste é conhecido como o *Teste do Meta-Interpretador* [55] e consiste em especializar um interpretador $\mathcal{M}$ para a própria linguagem adotada, em relação a programas P escritos nessa linguagem. O resultado deve ser o mais parecido possível com os programas P submetidos, se o avaliador parcial for poderoso o bastante para eliminar todo o processamento adicional associado à interpretação. Escrevemos um interpretador $\mathcal{M}$ para um subconjunto da linguagem adotada, compreendendo regras ASM básicas. O resultado da especialização de $\mathcal{M}$ em relação a especificações ASM foi muito bom, as especificações residuais produzidas são idênticas às fornecidas como entrada, exceto por renomeação de funções. Entretanto, para obter esses resultados, $\mathcal{M}$ utiliza a regra ASM não básica *var*. Não fomos capazes de expressar a interpretação dessa regra de modo a ainda obter bons resultados na especialização. Ou seja, $\mathcal{M}$ não é capaz de interpretar a si próprio.

Testes relacionados à compilação de programas, usando a primeira projeção de Futamura, produziram resultados satisfatórios. O teste mais significativo envolveu um interpretador para um subconjunto da linguagem C.

Para realizar experiências com a segunda projeção de Futamura, construímos uma versão do avaliador parcial escrita na própria linguagem ASM adotada, designado $\texttt{mix}_{ASM}$. Esse segundo avaliador parcial, entretanto, sofre um problema parecido com o enfrentado no desenvolvimento do meta-interpretador. Para obter bons resultados na especialização, foi necessária a utilização da regra ASM não básica *var*. Mas não fomos capazes de expressar a especialização dessa regra em $\texttt{mix}_{ASM}$ sem afetar negativamente a qualidade das especificações residuais.

Para geração de compiladores usando a segunda projeção de Futamura, o avaliador parcial original foi executado sobre $\texttt{mix}_{ASM}$, usando interpretadores como entrada estática. O resultado dos testes não foi muito animador, uma vez que não implementamos também muitas das otimizações necessárias em $\texttt{mix}_{ASM}$. A terceira projeção de Futamura não poderia ser testada, pois $\texttt{mix}_{ASM}$ não é capaz de processar seu próprio código.

Demonstramos formalmente diversas propriedades importantes sobre o funcionamento do avaliador parcial $\texttt{mix}_{ASM}$, incluindo a geração de especificações residuais corretas. Como $\texttt{mix}_{ASM}$ é escrito usando o próprio modelo ASM e uma linguagem funcional de avaliação

estrita, sua especificação e as demonstrações construídas são um exemplo de que o formalismo das Máquinas de Estado Abstratas é adequado para descrever um algoritmo de maneira clara e demonstrar com facilidade propriedades sobre o mesmo.

8.3 Extensões de Geração para ASM

Três foram os motivos que nos levaram a conduzir trabalhos envolvendo a abordagem de extensões de geração:

1. O desejo de investigar o maior número de técnicas diferentes associadas à especialização de programas.

2. A definição da linguagem Machĭna para realizar experimentos com o modelo ASM, incluindo experimentos com especialização. A abordagem de extensões de geração é mais adequada ao sistema de tipos de Machĭna do que a avaliação parcial tradicional.

3. Os problemas enfrentados com a auto-aplicação de um avaliador parcial tradicional para o modelo ASM. A abordagem de extensões de geração possibilita geração de compiladores sem os problemas associados à auto-aplicação.

Como o compilador de Machĭna não ficou pronto a tempo, utilizamos a linguagem Xasm em nossos experimentos. Desenvolvemos um gerador de extensões de geração `cogen` para o modelo ASM, com um núcleo de rotinas que não depende da linguagem concreta sendo utilizada. Um filtro que implemente uma interface entre o núcleo e programas da linguagem Machĭna pode ser facilmente desenvolvido, a exemplo do que foi feito para a linguagem Xasm.

A linguagem Xasm e seu compilador são atualmente o sistema mais confiável para execução de especificações ASM. O sistema gera código em C e depois gera um programa executável. Apesar de utilizar uma linguagem imperativa eficiente como linguagem objeto, consideramos que o compilador ainda assim não gera código muito eficiente. Na Seção 7.4.2, podemos observar algumas medidas de tempo para compilação e execução de especificações. Consideramos que o tempo para execução do interpretador de C escrito em Xasm, rodando programas C simples como ordenação de vetores, ainda é bastante alto. O objetivo principal de `cogen` é reduzir esse tempo de execução - os programas C compilados devem ser executados bem mais rapidamente que os interpretados.

O tempo que o gerador de extensões de geração `cogen` leva para gerar um compilador a partir de um interpretador escrito em Xasm é bastante razoável, bem melhor que o tempo gasto para gerar um executável com o compilador de Xasm. Vamos analisar então dois outros aspectos relacionados a `cogen`: o tempo de execução dos compiladores gerados e o tempo de excução dos programas compilados. O tempo de execução do compilador de C automaticamente gerado deixou a desejar, ao processar programas C um pouco mais longos. Uma melhoria média de 42% foi verificada, entre o tempo de execução dos programas compilados e dos programas interpretados. Isso ficou abaixo das nossas expectativas.

Consideramos que o ganho de eficiência obtido com a compilação, usando compiladores gerados por `cogen`, ainda não foi muito animador. Comparando o código Xasm dos programas compilados com o código do interpretador de C, podemos verificar que uma parte significativa das informações estáticas foi computada em tempo de especialização e não está presente nos programas residuais. Entretanto, o compilador Xasm gera um código pouco eficiente que é executado entre cada passo de uma execução de um programa ASM. Um grande número de testes e cópias de valores é realizado, mesmo que não seja necessário. Otimizações no compilador Xasm poderiam evitar isso. Assim, mesmo que um programa P_1 escrito em Xasm execute bem menos regras que um outro programa P_2 em um passo de suas execuções, a diferença percentual no tempo de execução não será muito acentuada.

Levando em conta as observações discutidas acima, consideramos que o código produzido pelo compilador C automaticamente gerado a partir de um interpretador é de boa qualidade. Outros experimentos foram conduzidos, envolvendo linguagens mais simples, por exemplo, um simulador da máquina de Turing. Os resultados obtidos foram semelhantes.

Esperamos que os compiladores produzidos por `cogen` a partir de interpretadores escritos na linguagem Machĭna executem de modo mais eficiente que os experimentos realizados com Xasm. O sistema de tipos de Machĭna deverá permitir uma geração de código executável mais eficiente, que foi o principal problema de Xasm. O código dos programas produzidos por esses compiladores automaticamente gerados terão os mesmos benefícios de eficiência.

Resumindo, podemos concluir que os experimentos com o gerador de extensões de geração `cogen` e a linguagem Xasm obtiveram bem mais sucesso que os experimentos com avaliação parcial tradicional. Compiladores foram gerados automaticamente a partir de interpretadores. A observação do código gerado por esses compiladores mostra que grande parte das informações estáticas dos interpretadores a partir dos quais foram gerados é corretamente eliminada dos programas residuais. Esses compiladores não executam de maneira muito eficiente, nem o ganho com o tempo de execução dos programas compilados é ainda muito significativo, mas esperamos obter mais sucesso com outra linguagem que segue o modelo ASM. O modo como cogen foi construído permite uma fácil adaptação para outra linguagem concreta.

8.4 Resumo das Contribuições da Pesquisa

Os resultados principais da pesquisa realizada podem ser resumidos como a seguir:

1. O desenvolvimento de técnicas e otimizações para avaliação parcial especificamente associadas a linguagens que seguem o modelo ASM. Essas técnicas foram utilizadas na implementação de um avaliador parcial para ASM, que obteve resultados satisfatórios no *teste do meta-interpretador*.

2. Demonstração formal da correção do avaliador parcial para ASM desenvolvido, usando o próprio modelo ASM e definições em uma linguagem funcional com avaliação estrita. Mostramos que o formalismo das Máquinas de Estado Abstratas é adequado

para descrever um algoritmo de maneira clara e demonstrar com facilidade propriedades sobre o mesmo.

3. A conclusão de que parece impossível especificar um avaliador parcial auto-aplicável usando o modelo ASM, com resultados satisfatórios para a especialização. Nosso trabalho levou a essa conclusão, mas não a demonstramos formalmente. Assim é algo que ainda pode ser mais investigado.

4. Adaptação das técnicas de avaliação parcial para ASM desenvolvidas, para a construção de um *gerador de extensões de geração* `cogen` para ASM. Essas técnicas foram utilizadas na implementação de `cogen` para ASM, com um núcleo de rotinas que pode ser usado com qualquer linguagem que siga o modelo ASM.

5. Desenvolvimento de uma interface para utilização da linguagem Xasm com o programa `cogen`. Realização de experimentos de geração automática de compiladores dirigida por semântica, usando Xasm e `cogen`. O principal experimento consistiu na geração de um compilador para um significativo subconjunto da linguagem C.

O nosso objetivo principal era a utilização de técnicas de avaliação parcial para tornar o uso do formalismo ASM mais atraente, com a geração automática de especificações especializadas, mais eficientes. Consideramos que os experimentos com o gerador de extensões de geração `cogen` mostraram que esse objetivo foi alcançado. Os resultados ficaram aquém das nossas expectativas em relação à eficiência, mas esperamos que interfaces com outras linguagens que sigam o modelo ASM poderão trazer melhorias de eficiência mais significativas.

8.5 Trabalhos Futuros

Um importante trabalho a ser desenvolvido, como discutimos nas seções anteriores, é uma integração de `cogen` com a linguagem Machĭna. Para isso, esperamos que o compilador de Machĭna para C esteja estável e gerando código eficiente. As modificações necessárias em `cogen` são simples, basta construir um novo filtro para Machĭna, seguindo os mesmos passos utilizados na construção do filtro para Xasm.

O interpretador para a linguagem C foi o exemplo mais significativo que utilizamos nos nossos experimentos. Esse interpretador ainda não processa toda a linguagem C, embora um significativo subconjunto tenha sido considerado. Particularmente, nos interessava observar a compilação de chamada de funções recursivas e a definição de variáveis em blocos. Nesse ponto, os resultados da compilação foram satisfatórios. O trabalho seguinte é estender o interpretador para cobrir todas as construções permitidas pela linguagem. Isso inclui alguns comandos de repetição, comandos `switch`, `goto`, `break`, vários operadores etc.

Na Seção 8.2, mencionamos a nossa incapacidade de descrever um avaliador parcial auto-aplicável para ASM que permitisse bons resultados na especialização, usando a segunda e terceira projeções de Futamura. Para realizar experimentos com a segunda projeção, fomos obrigados a especializar um avaliador parcial escrito em ASM usando

um outro avaliador parcial mais poderoso. Consideramos que ainda seja válida uma investigação que leve a concluir se é possível produzir um avaliador parcial auto-aplicável que satisfaça as exigências na especialização, ou então desenvolver uma demonstração que estabeleça os limites para isso.

Os nossos experimentos se concentraram na definição da semântica de linguagens de programação. Com o uso de técnicas de avaliação parcial, conseguimos resultados relacionados à geração automática de compiladores dirigida por semântica. Observamos melhorias no tempo de execução das especificações, mas não comparamos os resultados a outros sistemas de geração de compiladores dirigida por semântica. Uma comparação entre a aplicação de avaliação parcial junto com o modelo ASM e esses outros sistemas é um trabalho interessante a ser desenvolvido.

Finalmente, outro trabalho futuro consiste em aplicar as técnicas e especializadores desenvolvidos a especificações ASM relacionadas a outros campos de utilização, não relacionados à semântica de linguagens de programação. Infelizmente, não podemos ainda realizar experimentos com protocolos distribuídos escritos no modelo ASM, uma vez que consideramos apenas ASM seqüenciais. A pesquisa de técnicas de avaliação parcial para programas paralelos e distribuídos ainda não é muito desenvolvida. O modelo ASM poderá ser de grande ajuda nesse caso.

Referências Bibliográficas

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

[2] L. Andersen. C program specialization. Master's thesis, DIKU, University of Copenhagen, Denmark, December 1991. Student Project 91-12-17.

[3] L. Andersen. C program specialization. Technical Report 92/14, DIKU, University of Copenhagen, Denmark, May 1992.

[4] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[5] P. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[6] 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.

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

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

[9] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[10] R. S. Bigonha. The languague *script* for denotational semantics. Technical Report 05/1999, Laboratório de Linguagens de Programação, Universidade Federal de Minas Gerais, 1999.

[11] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, 1993. DIKU Research Report 93/22.

[12] L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 61–71, 1994.

[13] 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.

[14] L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3):191–208, September 1995.

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

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

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

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

[19] 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.

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

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

[22] 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.

[23] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.

[24] C. Consel and S. Khoo. Semantics-directed generation of a prolog compiler. Technical Report YALEU/DCS/RR-781, Yale University, New Haven, Connecticut, May 1990.

[25] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

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

[27] 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.

[28] 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.

[29] V. O. Di Iorio, R. S. Bigonha, and M. A. S. Bigonha. Tutorial em Avaliação Parcial de Programas. In *Anais do IV Simpósio Brasileiro de Linguagens de Programação*, Recife, Maio 2000.

[30] 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.

[31] V. O. Di Iorio, R. S. Bigonha, and M. M. Maia. A Self-Applicable Partial Evaluator for ASM. Technical Report LLP-11-99, Programming Languages Laboratory, DCC, Universidade Federal de Minas Gerais, 1999.

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

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

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

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

[36] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, Amsterdam: North-Holland, 1989.

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

[38] L. C. C. Guedes. *Um Modelo Orientado a Objetos para Geração Automática de Compiladores*. PhD thesis, PUC-Rio, Agosto 1995.

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

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

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

[42] 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.

[43] 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.

[44] 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.

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

[46] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Goedel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[47] J. Hatcliff, T. Mogensen, and P. Thiemann, editors. Partial Evaluation: Practice and Theory, volume 1706. Springer-Verlag, 1999.

[48] 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.

[49] C. Holst. Finiteness analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 473–495. ACM, Berlin: Springer-Verlag, 1991.

[50] C. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[51] J. Huggins. An Offline Partial Evaluator for Evolving Algebras. Technical Report CSE-TR-229-95, EECS Dept., University of Michigan, 1995.

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

[53] N. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. Amsterdam: North-Holland, 1988.

[54] N. Jones, C. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *1990 International Conference on Computer Languages, New Orleans, Louisiana, March 1990*, pages 49–58. New York: IEEE Computer Society, 1990.

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

[56] N. Jones and D. Schmidt. Compiler generation from denotational semantics. In N. Jones, editor, *Semantics-Directed Compiler Generation, Aarhus, Denmark (Lecture Notes in Computer Science, vol. 94)*, pages 70–93. Berlin: Springer-Verlag, 1980.

[57] 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.

[58] 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.

[59] S. L. P. Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.

[60] J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992. Student Project 92-1-4.

[61] 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.

[62] K. Kahn and M. Carlsson. The compilation of Prolog programs without the use of a Prolog compiler. In *International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 348–355. Tokyo: Ohmsha and Amsterdam: North-Holland, 1984.

[63] 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.

[64] P. Lee. *Realistic Compiler Generation*. MIT Press, Cambridge, MA, 1989.

[65] P. Lee and U. Pleban. A realistic compiler generator based on high-level semantics. In *Fourteenth Symposium on Principles of Programming Languages, Munich, Germany, January 1987*, pages 284–295. ACM: New York, 1987.

[66] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992. Available from `ftp.diku.dk` as file `pub/diku/semantics/papers/D-152.ps.Z`.

[67] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[68] T. Mogensen. Self-applicable online partial evaluation of pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*. New York: ACM, 1995.

[69] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *LOPSTR 92. Workshops in Computing*. Berlin: Springer-Verlag, Jan. 1993.

[70] T. Æ. Mogensen and P. Sestoft. Partial evaluation. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.

[71] P. Mosses. SIS — semantics implementation system, reference manual and user guide. DAIMI Report MD-30, DAIMI, University of Århus, Denmark, 1979.

[72] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Franciso, CA, USA, 1997.

[73] F. Nielson and H. R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56:59–133, 1988.

[74] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science **vol. 34**. Cambridge University Press, 1992.

[75] P. Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In P. Fritzon, editor, *Proceedings of the 1994 Conference on Compiler Construction, Edinburgh*, volume 786 of LNCS, pages 1–15. Springer-Verlag, April 1994. URL: `ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html`.

[76] U. Pleban. Compiler prototyping using formal semantics. In *Symposium on Compiler Construction (Sigplan Notices, vol. 19, no. 6, June 1984)*, pages 94–105. New York: ACM, 1984.

[77] S. Romanenko. Arity raiser and its use in program specialization. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990 (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Berlin: Springer-Verlag, 1990.

[78] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991. Report TRITA-TCS-9101.

[79] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. Amsterdam: North-Holland, 1988.

[80] P. Thiemann. Cogen in 6 lines. In *International Conference on Functional Programming (ICFP'97), Philadelphia, Pennsylvania, May 1996*, pages 180–189. New York: ACM, 1996.

[81] F. Tirelo, R. Bigonha, M. A. Maia, and V. Iorio. Machĭna: A Linguagem de Especificação de ASM. Technical Report 08/1999, Laboratório de Linguagens de Programação, Universidade Federal de Minas Gerais, 1999.

[82] F. Tirelo, R. Bigonha, M. A. Maia, and V. Iorio. Tutorial em Máquinas de Estado Abstratas. In *Anais do III Simpósio Brasileiro de Linguagens de Programação*, Porto Alegre, Maio 1999.

[83] M. Tofte. *Compiler Generators - What They Can Do, What They Might Do and What They Probably Never Do*, volume 19. Springer-Verlag, Mar. 1990.

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