Análise e Verificação de Programas

Leonardo Mendonça de Moura

Fevereiro de 2000

Meus agradecimentos:

- ao Professor Lucena por ter me orientado no desenvolvimento deste trabalho. E por ser compreensível em todos os momentos.
- ao Professor Hermann por ter me ajudado e apoiado no desenvolvimento deste trabalho.
- à minha esposa pelo apoio durante o curso de doutorado, tendo ficado várias dias me ajudando na redação e revisão deste trabalho.
- aos meus pais por me apoiarem na realização do curso de doutorado.
- ao Christiano Braga pela sua amizade e pela ajuda exaustiva na revisão do trabalho.
- ao Ira Baxter por ter sugerido o tema da tese e por suas observações contundentes.
- ao Christopher e Virginia Pidgeon por todo apoio provido a mim e a minha família durante o ano de 1998.
- ao Michael Mehlich pelas discussões que contribuíram para o desenvolvimento da tese.
- ao Marcelo Sant'Anna (Guru) pela amizade, conselhos e orientação espiritual.
- ao Laboratório de Engenharia de Software (LES), pelos recursos proporcionados por este, ajudando a tornar possível este trabalho.
- ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pela ajuda financeira recebida durante o curso.

Resumo

Atualmente, a análise de propriedades de programas é geralmente realizada por ferramentas específicas, que são desenvolvidas para cada linguagem de programação e propriedade de interesse. A construção manual destas ferramentas é dispendiosa e por isso estas são difíceis de serem obtidas e freqüentemente não são corretas, i.e. não concordam com a semântica da linguagem. Um argumento semelhante é válido para verificação de programas. A maioria das ferramentas de verificação é baseada em linguagens específicas que restringem a sua aplicação. Para verificarmos um programa utilizando estas ferramentas é necessário definirmos um mapeamento para a linguagem da ferramenta. Mas este tipo de mapeamento nem sempre é possível de ser realizado e/ou justificado, porque as linguagens utilizadas pelas ferramentas de verificação são em geral pouco expressivas. Assim, estamos propondo um "framework" conceitual para automatizar a construção de analisadores e verificadores de programas. A geração destas ferramentas é baseada na descrição formal da semântica operacional da linguagem em questão. Utilizamos também uma abordagem modular para a definição da semântica da linguagem, que possibilita a reutilização de módulos e fragmentos de ferramentas previamente definidas.

Abstract

Presently, program analysis and verification is at best achieved by a hand-crafted tool specific to a programming/specification language. Since the manual construction of tools is expensive, they are hard to obtain and not often used, limiting the quality of the implemented software. We have proposed a framework for the automated generation of simulation, analysis and verification tools for programs/specifications based on machine-processable formal definitions of the language's operational semantics. We also use a modular approach for language semantics that allows us to reuse semantic modules and fragments of already defined tools.

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Análise e Verificação de Programas

Análise e verificação de programas e especificações é um ramo importante da engenharia de software sendo utilizadas na otimização de código gerado por compiladores, na detecção de erros em programas e especificações, e na transformação de software em processos de engenharia e reengenharia. *Análise de programas* é definida como a geração de informação sobre o comportamento dinâmico de programas através da análise estática do código. Isto é, o programa não é executado explicitamente com todas as possíveis entradas para obter o resultado da análise. Nesta tese, definimos, de forma pragmática, que *verificação* é o processo de detecção de erros em programas, diferentemente da definição "clássica" de *verificação*, cujo o propósito é "provar a correção" de programas.

Atualmente, a análise de propriedades de programas é geralmente realizada por ferramentas específicas, que são desenvolvidas para cada linguagem de programação e propriedade de interesse. A construção manual destas ferramentas é dispendiosa e por isso estas são difíceis de serem obtidas e freqüentemente não são corretas, i.e. não concordam com a semântica da linguagem. Um argumento semelhante é válido para verificação de programas. A maioria das ferramentas de verificação é baseada em linguagens específicas que restringem a sua aplicação. Para verificarmos um programa utilizando estas ferramentas é necessário definirmos um mapeamento para a linguagem da ferramenta. Mas este tipo de mapeamento nem sempre é possível de ser realizado e/ou justificado, porque as linguagens utilizadas pelas ferramentas de verificação são em geral pouco expressivas.

Engenheiros de software se interessam em propriedades que nem sempre são suportadas por um conjunto limitado de analisadores específicos pré-desenvolvidos. Desta forma, é desejável que o engenheiro possa desenvolver novos analisadores de forma modular reutilizando analisadores ou partes destes pré-existentes.

Nós temos uma forte crença de que a construção de analisadores e verificadores de programas deve possuir uma base teórica que forneça ferramental suficiente, matemático ou não, para justificar a sua construção; seja realizada em um alto nível de abstração, tomando como base a semântica formal da linguagem; e seja o mais modular possível, a fim de possibilitar a reutilização dos analisadores.

Assim, estamos propondo um "framework" conceitual [1] para a construção de analisadores e verificadores de programas. Este "framework" é baseado nas teorias de *Interpretação Abstrata* [27, 25] e *Verificação de Modelos* (*Model Checking*) [19, 20] [2]. A teoria de Interpretação

[1] O uso da palavra "framework" não está sugerindo uma relação com o significado desta palavra na literatura de orientação a objetos.

[2] Um módulo de *Inclusion Constraint Solving* também está disponível, mas este não é parte significativa de

Abstrata fornece um método para construir descrições semânticas aproximadas de programas. Estas descrições aproximadas podem ser utilizadas para coletar informações sobre programas com o intuito de produzir respostas consistentes (*safe*) para perguntas sobre o comportamento dos programas em tempo de execução. Estas aproximações também podem ser utilizadas no processo de verificação de software. No caso de analisadores automáticos as respostas são aproximadas, já que questões sobre terminação são indecidíveis. A teoria de *Verificação de Modelos* colabora com diversas técnicas de verificação de programas concorrentes e linguagens [3] mais sofisticadas para descrição de propriedades e assertivas em programas. Apesar destas duas teorias terem sido desenvolvidas de formas independentes, acreditamos que sejam complementares e possuam diversas similaridades, que permitiram o desenvolvimento de um "framework" comum para análise e verificação de programas.

Durante o desenvolvimento do "framework" também criamos novas técnicas de análise e verificação de programas. Em especial, foi desenvolvida um técnica para análise de programas concorrentes. É importante salientar, que várias destas técnicas podem ser utilizadas independentemente do nosso "framework". Um prótipo para validação do "framework" e das idéias propostas foi implementado.

Apesar de nosso "framework" poder ser utilizado no desenvolvimento de analisadores e verificadores de código para diferentes linguagens de programação, estamos particularmente interessados no tratamento de linguagens específicas de domínio (*DSL - Domain Specific Language*) [10, 48], visto que acreditamos que o uso de linguagens simples e expressivas dentro de um domínio específico é um caminho interessante a ser seguido no desenvolvimento de software. Dentro deste escopo mostraremos como nosso "framework" pode ser utilizado para fazer verificação, transformação e otimização de programas [4] escritos em DSLs.

1.2 Problemas de decidibilidade

O *teorema de Rice* (Apêndice A), derivado do "*problema da parada*", diz que verificar propriedades de programas é indecidível. O "*problema de parada*" diz que é indecidível verificar se um programa termina. A demonstração deste resultado "clássico" é extremamente simples. Suponha que exista um programa *Halt(prog,input)* que verifica se o programa *prog* termina ao receber a entrada *input*. A partir de *Halt*, definimos o programa *Confunde* descrito na Figura 1.1. Se *Confunde(Confunde)* termina, então *Halt(Confunde, Confunde)* retorna *true* e *Confunde(Confunde)* não termina. Se *Confunde(Confunde)* não termina, então *Halt(Confunde, Confunde)* retorna *false* e *Confunde(Confunde)* termina. Como em ambos os casos obtivemos um paradoxo, a nossa suposição não é válida, e portanto o programa *Halt* não existe. Note que esta demonstração é semelhante ao paradoxo do mentiroso "*Eu estou mentindo.*" ou "*Esta frase é falsa.*".

A partir desta demonstração fica claro, também, que o problema está diretamente relacionado a programas que recebem outros programas como dados. Infelizmente este é exatamente o caso dos nossos analisadores e verificadores. De qualquer forma, esta situação não é tão desesperadora quanto parece. O "*problema da parada*" apenas garante que é impossível construir um programa que verifique se *qualquer* programa termina. O teorema não restringe a existência de programas que verifiquem se uma determinada classe de programas termina. Além disso, a demonstração do teorema está dentro de um contexto onde as *máquinas* que executam os programas possuem uma quantidade *potencialmente* infinita de memória ou recursos.

nosso "framework".

[3] Lógicas modais.

[4] Programas escritos em uma *DSL* também podem ser vistos como especificações.

```
Confunde(prog)
{
  if Halt(prog,prog)
    while (true) skip; /* loop */
  else
    return;
}
```

Figura 1.1: Programa *Confunde*

O "*problema da parada*" pode ser analisado, também, do ponto de vista da engenharia. Neste caso, as máquinas possuem quantidades finitas de memória e recursos, e podem, portanto, assumir apenas uma quantidade finita de estados. Conseqüentemente, os programas executados nestas máquinas podem apenas assumir uma quantidade finita de estados diferentes. Como a quantidade de estados que um programa assume é finita, podemos verificar qualquer propriedade desse programa simplesmente gerando todos os possíveis estados que o programa assume. Por exemplo, verificar terminação seria reduzido a procurar ciclos. Denominaremos $\mathcal{V}$ o programa que gera e armazena todos os estados que um programa $\mathcal{X}$ assume com o intuito de verificar propriedades. Entretanto, esta solução contém um problema de engenharia, porque o programa $\mathcal{V}$, em geral, consome exponencialmente mais memória que o programa $\mathcal{X}$. Assim, para analisarmos um programa $\mathcal{X}$ que é executado em uma máquina com uma quantidade m de memória, precisaremos executar o programa $\mathcal{V}$ em uma máquina com quantidade $O(e^m)$ de memória. Portanto, este programa $\mathcal{V}$ está limitado a analisar programas que utilizem um quantidade de memória muito menor que a utilizada por ele mesmo. A classe de programas que $\mathcal{V}$ consegue verificar está bem definida. O programa *Confunde* utilizado na demonstração do "problema da parada" explora esse fato. O programa *Confunde* é mais complexo que o programa *Halt*, porque este invoca *Halt*. O programa $\mathcal{V}$ é extremamente "ingênuo" como verificador, e além disso somente verifica propriedades módulo quantidade de recursos disponível. Entretanto, todas as técnicas de análise e verificação apresentadas podem ser consideradas como versões sofisticadas do programa $\mathcal{V}$, já que:

- utilizam aproximações para lidar com a complexidade de programas;
- exploram regularidades nos estados assumidos por um programa;
- utilizam técnicas mais sofisticadas de coleta e verificação de propriedades;
- não assumem, em geral, uma quantidade finita de recursos [5].

1.3 Correção de Software?

> *"Beware of bugs in the above code; I have only proved it correct, not tried it"*
> Knuth [57]

[5] A motivação para este ítem é puramente prática, já que na maioria das vezes estamos interessados em mostrar que um programa ou algoritmo possui determinada propriedade independentemente da quantidade de memória recursos utilizada para executá-lo.

É muito comum encontrarmos argumentos sobre a correção do software $\mathcal{X}$ ou $\mathcal{Y}$. Argumentos sobre correção envolvem um programa e uma especificação, i.e. uma descrição do programa em alto nível da abstração. Podemos dizer, neste caso, que o software $\mathcal{X}$ está correto em relação a especificação $\mathcal{S}$, i.e. ele satisfaz a especificação. Contudo como iremos garantir que uma especificação esteja correta? Uma maneira é tentar verificar que uma dada especificação possui certas propriedades. É claro que, este processo não garante que uma especificação esteja correta, este no máximo mostra que a especificação esteja incorreta, ou que haja algum erro em nossas suposições.

Dentro do nosso ponto de vista é completamente "utópico" atingir a "correção" de software. Acreditamos, então, que no desenvolvimento de software o desenvolvedor está em um processo contínuo de "auto-convencimento" de que o software está "correto". Podemos, então, fazer um paralelo com o princípio de *plausible reasoning* descrito por Polya [79]. Dentro deste princípio, temos que se "$A \Rightarrow B$" e "*B é verdadeiro*", então "*A é mais digno de crédito*". Este princípio pode ser traduzido para as ciências naturais como: "*Teoria A prevê* ($\Rightarrow$) o *fenômeno B*" e "*B é observado*", então "*A é mais digno de crédito (B é uma evidência da Teoria A)*". Este princípio também pode ser traduzido para o desenvolvimento de software como: "*Programa/Especificação A deve possuir* ($\Rightarrow$) *a propriedade B*" e "*B é verificada*" (ou "*a ausência de B não é detectada*"), então "*A é mais digno de crédito*". Assim, o desenvolvedor aumentou a sua confiança de que o software faz o que deveria fazer.

Dentro deste princípio, basta uma observação inválida para descartarmos ou modificarmos uma teoria ou programa.

No desenvolvimento de software existem diversas formas de aumentarmos nossa confiança em que um determinado programa esteja "correto".

- Execução de um programa em um caso particular. Por exemplo, o programa *rqd* (raiz quadrada) deve possuir a propriedade $rqd(4) = 2$.
- Uso de analisadores que extraem propriedades envolvendo diversas (ou todas) execuções do programa. Por exemplo, o programa *max* (valor máximo) deve possuir a propriedade $\forall a, b \cdot max(a, b) \geq a$.
- Uso de verificadores que *tentam* detectar erros, usando aproximações ou não, nos estados que um programa assume. Como, em geral, a quantidade de estados que um programa assume é potencialmente infinita, os verificadores não garantem a ausência do erro.

É claro que, os analisadores e verificadores produzem evidências mais fortes do que a execução de um programa em casos particulares.

Pode parecer irônico, mas um conjunto "infinito" de evidências não garante que um programa esteja "correto", mas apenas uma observação inválida garante que ele está incorreto.

> *"The only sensible goal of formal methods is to detect the presence of errors"* Henzinger [45]

1.4 Usando analisadores e verificadores

Em nosso "framework", analisadores e verificadores de código são especificados em um linguagem de programação denominada *PAN*. Esta linguagem lógica possui primitivas que permitem a construção modular deste tipo de ferramentas. Como intepretadores de programas podem ser vistos como um caso particular de analisadores e verificadores, o nosso "framework" também pode ser utilizado na construção destes.

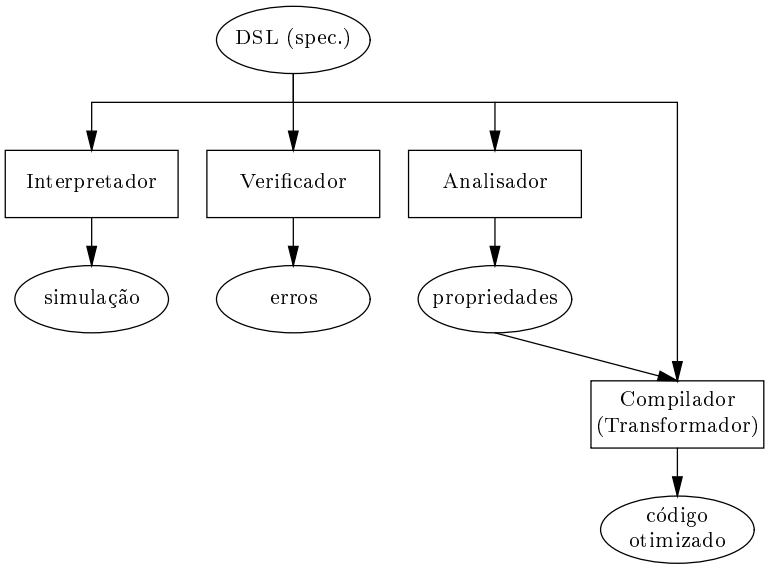

Figura 1.2: Analisadores e Verificadores no processo de desenvolvimento de software

A Figura 1.2 descreve a função de cada uma destas ferramentas. Nesta figura, estamos considerando a aplicação de nosso "framework" a uma *DSL*. *DSLs* podem ser vistas como especificações, devido ao seu alto nível de abstração. Dentro deste contexto, temos que:

- os interpretadores podem ser utilizados para realizar simulações sobre o programa/especificação;
- os verificadores são utilizados para detectar erros através da exploração exaustiva dos estados que o programa/especificação assume;
- os analisadores são utilizados para extrair propriedades que podem ser utilizadas para aumentar a nossa confiança que o programa/especificação está correta ou para transformá-lo (i.e. compilá-lo) em código eficiente.

É importante salientar, que nesta tese estamos apenas considerando a verificação e análise de linguagens que possuam uma interpretação operacional. Todas as ferramentas produzidas em nosso "framework" são baseadas na noção de transição de estado.

1.5 Estrutura da Tese

Nos capítulos 2, 3 e 4, apresentaremos o fundamento teórico do nosso "framework" de análise e verificação de programas. No capítulo 5 descreveremos a estrutura geral dos analisadores e verificadores do nosso "framework". No capítulo 6 será apresentada a linguagem *PAN*, que é utilizada para especificar os analisadores de programa dentro do nosso "framework". O capítulo 7 conterá os detalhes de implementação da linguagem *PAN*. Os capítulos 8 e 9 mostrarão como analisadores e verificadores de programas podem ser implementados em nosso "framework". O capítulo 9 também contem diversos exemplos de analisadores e verificadores, que construímos utilizando as técnicas descritas nesta tese. No capítulo 10 faremos uma comparação com trabalhos correlatos.

Para evitarmos a repetição excessiva do termo "analisadores e verificadores", optamos pela utilização do termo "analisadores" para designar os dois tipos de ferramenta, sempre que tal uso não gere uma ambigüidade.

Capítulo 2

Interpretação Abstrata

2.1 Introdução

Interpretação Abstrata [27, 25] é um método para construir descrições semânticas aproximadas de programas. Uma descrição semântica aproximada pode ser utilizada para coletar propriedades de programas com o objetivo de prover respostas seguras sobre o comportamento dinâmico dos programas. Estas aproximações podem ser utilizadas, também, no processo de verificação de software. As respostas providas são parciais ou aproximadas, já que questões sobre terminação são indecidíveis. A teoria de Interpretação Abstrata fornece o ferramental necessário para justificarmos a correção das descrições aproximadas em relação a semântica da linguagem.

Do ponto de vista teórico, a teoria de Interpretação Abstrata tem como propósito a definição de hierarquias de descrições semânticas que especificam o comportamento do programa em diferentes níveis de abstração. A partir deste ponto de vista, podemos ver a palavra "interpretar" como "explicar o significado de" e "abstrata" como "entender de forma específica". Do ponto de vista prático, podemos ver "interpretar" como "agir como um interpretador" e "abstrata" como "trocar os domínios semânticos concretos por domínios aproximados".

A exposição neste capítulo sobre Interpretação Abstrata já apresenta influências da teoria de Verificação de Modelos [93, 86, 30] envolvendo explicitamente conceitos como *traces* e mapas de estados. Entretanto, sempre que possível, mostraremos os contrastes e diferenças em relação a teoria de Interpretação Abstrata "clássica" [27] e a literatura de *análise de fluxo de dados*[1, 68].

2.2 Mapa de Estados

A semântica operacional de uma linguagem de programação define o comportamento dos programas em tempo de execução. Denominaremos esta de *semântica concreta* ou *interpretação concreta*. Como a semântica operacional pode ser vista como a definição de um interpretador, podemos executar um programa com os seus dados de entrada. Porém, não estamos somente interessados no valor retornado pela computação, e sim por todos os estados ou configurações que a computação passar. O encadeamento cronológico destes estados será denominado *mapa de estados*. É natural acreditar que o mapa de estados nada mais é do que uma seqüência de estados. Mas, como a semântica operacional de uma linguagem nem sempre é determinística [1], o mapa de estados pode também ser uma árvore que representa todas as possíveis execuções do programa. Neste caso, um caminho a partir da raiz da árvore é denominado *trace* de execução. Quando uma computação divergir, teremos *traces* infinitos de execução.

Como exemplo, a Figura 2.1 contém um programa simples, é importante ressaltar que este

[1] Linguagens com suporte a concorrência e paralelismo não são determinísticas.

```
INPUT x;
y = 1;
L1:
while (x > 1)
{ L2: y = y * x;
  L3: x = x - 1;
  L4:
}
L5:
```

Figura 2.1: Programa Fatorial

programa possui vários *labels* cuja a única função é facilitar a nossa exposição do mapa de estados. A Figura 2.2 contém o mapa de estados deste programa considerando a entrada $x = 3$.

Ao substituirmos os valores que as variáveis (dados) do programa assumem por valores abstratos e adaptarmos a semântica a essas abstrações, produziremos um mapa de estados abstratos. As abstrações representam propriedades dos dados concretos do programa. O processo de construir o mapa de estados abstratos será denominado de *interpretação abstrata*. É conveniente vermos interpretação abstrata como um tipo de "execução simbólica". Como exemplo, apresentaremos um analisador de sinais, onde valores inteiros como 1 e -3 são substituídos pelas versões abstratas *positivo* e *negativo* respectivamente. Outro exemplo, é a inferência de tipos [26], onde os dados concretos (ex.: *false* e 3) são substituídos pelos respectivos tipos (ex.: *boolean* e *integer*).

Quando os dados concretos são substituídos por versões abstratas, a semântica da linguagem deve ser modificada de forma apropriada. Por exemplo, o operador $\hat{+}$ abstrato deve se comportar de forma consistente com o operador + concreto. Como $2+3=5$, temos que *positivo* $\hat{+}$ *positivo* = *positivo*.

Interpretações Abstratas são extremamente não determinísticas, mesmo que o programa que esteja sendo interpretado seja determinístico. O não determinismo é proveniente do uso de abstrações. Por exemplo, o comando

$$\text{if } (x > 10)\ x = x + 1$$

não pode ser executado deterministicamente se durante a interpretação abstrata o valor da variável x for *positivo*. Porém, se o valor for *negativo* com certeza o corpo do comando *if* não precisará ser executado.

A abordagem "clássica" de interpretações abstratas não faz referência explícita ao conceito de mapa de estados. Entretanto, dentro do nosso ponto de vista, o *diagrama de fluxo de controle* [27, 1, 68] possui um propósito semelhante, porque este diagrama é utilizado em conjunto com um cálculo de ponto fixo para extrair propriedades de programas. Outra semelhança entre a duas abordagens diz respeito a noção de transição de estado, que é modelada na abordagem "clássica" através de *funções de transferência*. Estas funções tem como objetivo abstrair o efeito de um comando ou bloco de código (i.e. *basic blocks* [2]).

[2]Formalmente, um *basic block* é um seqüência maximal de instruções (comandos) cuja a execução pode ser iniciada apenas no primeiro elemento, e terminada no último. Isto é, não pode ocorrer desvios durante a execução de um *basic block*.

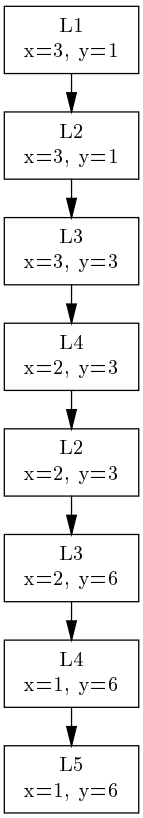

Figura 2.2: Mapa de estados do programa fatorial com entrada x = 3

2.3 Exemplo: Analisador de Sinais

Um dos exemplos mais simples de interpretação abstrata é o analisador de sinais. Este analisador tenta inferir o sinal das variáveis de um programa. Os valores abstratos (propriedades) estão organizados no reticulado descrito na Figura 2.3. Neste reticulado, o elemento $\top$ (*top*) representa a propriedade "não sei", assim este é utilizado sempre que não conseguirmos extrair precisamente o sinal de uma variável. A Figura 2.4 contém o mapeamento entre os valores concretos e abstratos. As Figuras 2.5 e 2.6 contêm as definições dos operadores abstratos de soma e multiplicação.

Ao considerarmos, mais uma vez, como exemplo o programa descrito na Figura 2.1, a Figura 2.7 contém o mapa de estados abstratos associado ao programa. Este exemplo ilustra o uso de grafos para representar árvores infinitas. Neste exemplo, o mapa de estados é exatamente igual a um diagrama de fluxo de controle decorado com o valores obtidos através da análise de fluxo de dados. Entretanto, esta equivalência nem sempre é verdadeira. Por exemplo, considere o programa descrito na Figura 2.8 e um reticulado de sinais um pouco mais "rico" [3] (Figura 2.9). Supondo que o corpo dos comandos *if* não modifiquem o valor da variável x, obteremos o mapa de estados abstratos descrito na Figura 2.10, que é diferente do diagrama de fluxo de controle do programa (Figura 2.11). Além disso, se utilizarmos este diagrama de fluxo de controle como base para a coleta de propriedades do programa, estaremos considerando o caminho (L1, L4, L5, L6, L7, L8, L9), que é impossível de ser executado. Ao considerarmos caminhos impossíveis de serem executados, estaremos possivelmente prejudicando o resultado de nossa análise. Portanto, a nossa abordagem baseada em mapas de estados é mais precisa que a abordagem "clássica".

[3] Este é o reticulado normalmente usando na prática.

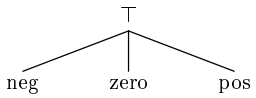

Figura 2.3: Diagrama de Hasse do reticulado de sinais

$$neg \leftrightarrow \{-1, -2, -3, \ldots\}$$
$$zero \leftrightarrow \{0\}$$
$$pos \leftrightarrow \{1, 2, 3, \ldots\}$$
$$\top \leftrightarrow \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$$

Figura 2.4: Definição dos elementos do reticulado de sinais

$\hat{\times}$	*neg*	*zero*	*pos*	$\top$
neg	pos	zero	neg	$\top$
zero	zero	zero	zero	zero
pos	neg	zero	pos	$\top$
$\top$	$\top$	zero	$\top$	$\top$

Figura 2.5: Definição do operador abstrato de multiplicação

$\hat{+}$	*neg*	*zero*	*pos*	$\top$
neg	neg	neg	$\top$	$\top$
zero	neg	zero	pos	$\top$
pos	$\top$	pos	pos	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Figura 2.6: Definição do operador abstrato de soma

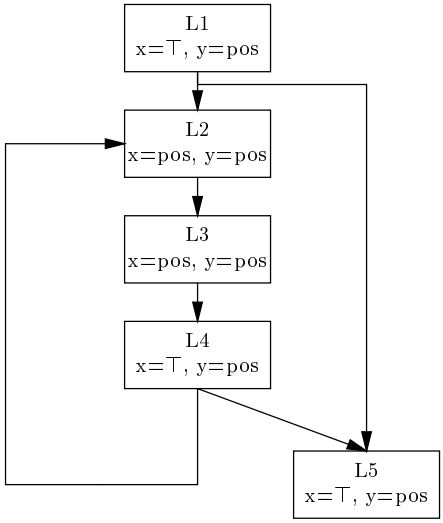

Figura 2.7: Mapa de estados abstratos do programa fatorial

```
INPUT x;
L1:
if (x >= 0)
{ L2: ... L3: }
else
{ L4: ... L5: }
L6:
if (x > 10)
{ L7: ... L8: }
L9:
```

Figura 2.8: Programa exemplo *2-ifs*

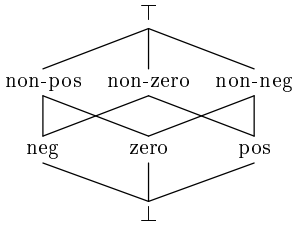

Figura 2.9: Diagrama de Hasse do reticulado de sinais extendido

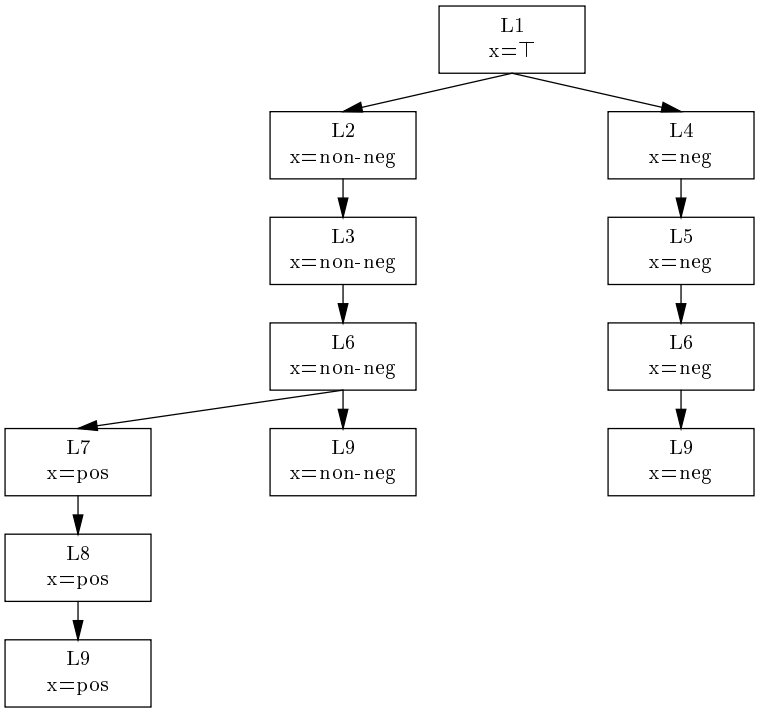

Figura 2.10: Mapa de estados abstratos do programa *2-ifs*

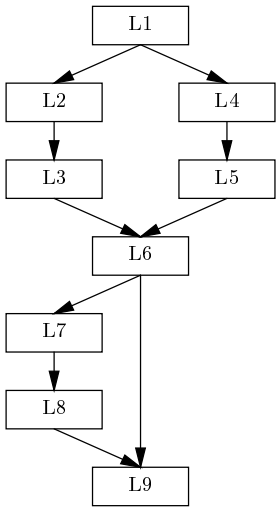

Figura 2.11: Fluxo de controle do programa *2-ifs*

2.4 Ordens Parciais e Reticulados

No exemplo do analisador de sinais, apresentamos dois reticulados que continham os elementos abstratos (propriedades) que eram utilizados durante a análise. Nesta seção definiremos os conceitos de ordens parciais (*poset*) e reticulados. Basicamente todo o conjunto de valores abstratos utilizado em nossos analisadores possui uma noção de ordem parcial, que iremos representar através do símbolo $\sqsubseteq$. Esta ordem captura a noção de um valor ser uma aproximação melhor do que outro [28]. Por exemplo, $pos \sqsubseteq \top$ e $neg \sqsubseteq non\text{-}pos$. A nossa noção de ordem é parcial, visto que existem elementos incomparáveis (ex.: *pos* e *neg*). Podemos, também, ver esta noção de ordem como uma forma de relacionar quantidade de informação, assim teremos $a \sqsubseteq b$ se a contém menos informação que b. Neste caso, podemos usar um mapeamento como o descrito na Figura 2.4 para reduzirmos a definição do operador $\sqsubseteq$ a do operador $\subseteq$ entre conjuntos. Assim temos que $a \sqsubseteq b$ se $set(a) \subseteq set(b)$.

Para construirmos um analisador, e justificarmos a sua correção, precisamos no mínimo de uma noção de ordem para o conjunto de valores abstratos [4]. Entretanto, para alguns analisadores precisamos de mais estrutura. Esta estrutural adicional é necessária, em geral, para garantir a terminação do analisador. Nestes casos é muito comum utilizarmos reticulados. Um reticulado nada mais é do que um conjunto L com dois operadores $\sqcup$ (*join*) e $\sqcap$ (*meet*) que possuem as seguintes propriedades:

1. $\forall x, y \in L \ \exists z, w \in L \cdot x \sqcup y = z$ e $x \sqcap y = w$
2. $\forall x, y \in L \cdot x \sqcup y = y \sqcup x$ e $x \sqcap y = y \sqcap x$
3. $\forall x, y, z \in L \cdot x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$ e $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$
4. Existem dois elementos em L denominados *bottom*, representado por $\bot$, e *top*, representado por $\top$, tal que $\forall x \in L \cdot x \sqcap \bot = \bot$ e $x \sqcup \top = \top$

A definição de reticulado deixa claro porque o $\top$ pode ser visto como a resposta "não sei". Nesta definição $\top$ é o elemento maximal do reticulado. Dentro do nosso contexto de analisadores $\top$ é, então, a pior aproximação possível, já que é o elemento que contém mais informação.

[4] O conjunto de valores abstratos precisa ser uma ordem parcial.

Muitos reticulados também são distributivos, i.e.,

$$\forall x, y, z \in L \cdot (x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z) \text{ e } (x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z)$$

Os operadores de *meet* e *join* induzem uma ordem parcial $\sqsubseteq$, i.e., todo reticulado também é uma ordem parcial. O operador $\sqsubseteq$ pode ser definido em função do *meet* da seguinte forma:

$$x \sqsubseteq y \Leftrightarrow x \sqcap y = x$$

Um reticulado é dito completo, se para todo $S \subseteq L$, existe $x = \sqcup\{s \in S\}$ (o *join* de um conjunto de elementos), e $y = \sqcap\{s \in S\}$ (o *meet* de um conjunto de elementos). Estes valores sempre existem se S for um subconjunto finito, mas a existência destes valores não é garantida, em qualquer reticulado, se o subconjunto é infinito.

A partir da definição de *ordens parciais* e *reticulados* podemos definir o conceito de função monótona. Uma função $f : L \to L$ é monótona se $\forall x, y \in L \cdot x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

A altura de um *poset* ou *reticulado* é o comprimento da maior cadeia estritamente crescente:

$$\bot = x_1 \sqsubset x_2 \sqsubset \ldots \sqsubset x_n = \top$$

Onde $x \sqsubset y$ se $x \sqsubseteq y \wedge x \neq y$. Se não existe n tal que $x_n = \top$, dizemos que a altura é infinita. É importante notar que para esta definição de altura se aplicar a ordens parciais, estas devem possuir um elemento máximo $\top$. Esta condição será sempre satisfeita por nossos analisadores visto que, $\top$ representa a resposta "não sei". Desta forma, os nossos conjuntos de valores abstratos devem ser pelo menos ordens parciais que possuam um elemento maximal $\top$.

2.5 Relacionando Mapas de Estados Concretos e Abstratos

Intuitivamente um mapa de estados abstratos está relacionado a um mapa de estados concretos através de um homomorfismo. Então, seja $\alpha : \mathit{Val} \to \mathit{AbsVal}$, uma função de abstração que leva um valor concreto no valor abstrato que melhor o representa. Por exemplo, no analisador de sinais temos que $\alpha(1) = \mathit{pos}$ e $\alpha(-3) = \mathit{neg}$. De posse da função α de abstração de *valor*, podemos definir uma função α_{state} de abstração de estado. Nesta definição, estamos assumindo que a memória (*store*) é modelada como um conjunto de pares (*variável, valor*) e que o estado é representado pelo par (*control-point, store*). Um *control-point* especifica uma localização do programa. Também podemos ver um *control-point* como o valor contido no *instruction-pointer* de um interpretador, que especifica o próximo comando a ser executado.

$$\alpha_{state}(\textit{control-point}, \textit{store}) = (\textit{control-point}, \alpha_{store}(\textit{store}))$$

$$\alpha_{store}(s) = \{(x, \alpha(val)) | (x, val) \in s\}$$

Independentemente da definição da função de abstração de estados, podemos definir o homomorfismo entre um mapa de estados concretos e um mapa de estados abstratos da seguinte forma:

$$\forall s, s' \cdot s \to s' \Rightarrow \exists a' \cdot \alpha_{state}(s) \to a' \text{ e } \alpha_{state}(s') \sqsubseteq a'$$

Este relacionamento entre os mapas pode ser entendido da seguinte forma: para toda transição concreta $s \to s'$, existe um estado abstrato a' e uma transição abstrata entre a abstração do estado s e a' ($\alpha_{state}(s) \to a'$), tal que, o estado abstrato a' contém mais informação do que a abstração do estado s'. Ou seja, a' é uma aproximação de s' ($\alpha_{state}(s') \sqsubseteq a'$), mas não necessariamente a melhor de todas, que seria o caso de $\alpha_{state}(s') = a'$.

Tendo definido o homomorfismo entre os mapas de estados, temos que definir uma noção de "segurança", ou seja, que todos os comportamentos produzidos pela interpretação concreta possam ser "simulados" pela interpretação abstrata. Para isso, definimos a relação $safe \subseteq Val \times AbsVal$ [87]

$$c \; safe \; a \Leftrightarrow \alpha(c) \sqsubseteq a$$

Neste caso, dizemos que c é aproximado com "segurança" por a [74]. A partir dessa definição de "segurança" podemos definir uma relação semelhante entre estados concretos e abstratos, usando a definição α_{state}, temos que:

$$cs \; safe_{state} \; as \Leftrightarrow \alpha_{state}(cs) \sqsubseteq as$$

Todos os mapas de estados são na verdade árvores finitas ou infinitas. No caso de árvores infinitas que possuam alguma regularidade (Figura 2.7), grafos podem ser utilizados como uma forma finita de representação. Logo, para definirmos uma relação de segurança entre mapas de estados, devemos definir uma noção de segurança entre árvores. Para um mapa de estados (árvore) t, escrevemos $root(t)$ para denotar a raiz da árvore t, e escrevemos $t \rightarrow t'$ para denotar que existe uma transição entre a raiz de t e a raiz de t', i.e., $root(t) \rightarrow root(t')$. A partir dessas definições, dizemos que um mapa de estados (árvore) concretos é aproximado seguramente por um mapa de estados abstratos se:

$$\begin{aligned} t \; safe_{tree} \; t' \quad \Leftrightarrow \quad & root(t) \; safe_{state} root(t'), \text{ e para toda transição, } t \rightarrow t_i, \\ & \text{existe uma transição, } t' \rightarrow t'_j, \text{tal que, } t_i \; safe_{tree} \; t'_j \end{aligned}$$

Intuitivamente esta relação diz que todo caminho (*path*) na árvore (mapa de estados) concreta é simulado com segurança por uma caminho na árvore abstrata. É importante observar que a definição é co-recursiva, sendo necessário usarmos argumentos envolvendo co-indução, já que estamos lidando com árvores finitas e infinitas, e estamos interessados no maior ponto fixo desta definição [5]. Uma introdução a co-indução pode ser encontrada no Apêndice D. A necessidade de utilizarmos co-indução não é uma surpresa, visto que, a definição de $safe_{tree}$ é bastante semelhante com a definição de *Bissimilaridade* [75, 63] (Seção D.1) utilizada em equivalência observacional. Por conseguinte, podemos dizer que $safe_{tree}$ é uma forma "fraca" de equivalência.

A partir do momento que utilizamos essas definições, temos que dado um mapa de estados concretos, um mapa de estados abstratos é uma simulação segura se a seguinte propriedade for válida:

$$\forall s, s' \forall a \cdot s \rightarrow s' \text{ e } s \; safe_{state} \; a \Rightarrow \exists a' \cdot a \rightarrow a' \text{ e } s' \; safe_{state} \; a'$$

Esta propriedade pode ser facilmente demonstrada equivalente ao homomorfismo definido anteriormente. Esta propriedade garante que a simulação é segura se $safe_{state}$ for *U-fechado*, isto é, $s \; safe_{state} \; a \wedge a \sqsubseteq a' \Rightarrow s \; safe_{state} \; a'$ e a relação de transição abstrata for monótona, isto é:

$$\forall a_1, a'_1, a_2 \cdot a_1 \rightarrow a'_1 \text{ e } a_1 \sqsubseteq a_2 \Rightarrow \exists a'_2 \cdot a_2 \rightarrow a'_2 \text{ e } a'_1 \sqsubseteq a'_2$$

Teorema 1 *Dada uma simulação (i.e. um mapa de estados abstratos), onde a relação* $\text{safe}_{\text{state}}$ *é U-*fechada*, a relação de transição abstrata é monótona, e vale a seguinte propriedade:*

$$\forall s, s' \forall a \cdot s \rightarrow s' \; e \; s \; safe_{state} \; a \Rightarrow \exists a' \cdot a \rightarrow a' \; e \; s' \; safe_{state} \; a'$$

então, a simulação é segura

[5] O menor ponto fixo desta definição é a relação vazia, sendo, portanto, de pouca utilidade.

Em geral, é trivial mostrar que a relação de transição abstrata é monótona e que a relação $safe_{state}$ é *U-fechado*. Desta maneira, para mostrarmos a segurança de uma simulação (i.e. de um mapa de estados abstratos), temos que mostrar que dado um estado concreto s e uma aproximação a deste estado, para todas as transições $s \to s'$ existe uma transição $a \to a'$ tal que a' é uma aproximação de s'.

Na abordagem "clássica" a segurança dos resultados é garantida através da observação que *post fixpoints* são aproximações do menor ponto fixo, e do conceito de *Conexão de Galois*, como mostraremos a seguir.

Teorema 2 *Se* $(L, \sqsubseteq)$ *é um* poset, $F \in L \to L$ *é uma função contínua, e* $\exists a \in L \cdot F(a) \sqsubseteq a$, *então* $\mathrm{lfp}(F) \sqsubseteq a$. *Isto é, se* a *é um* post fixpoint *de* F, *então* a *é uma aproximação do menor ponto fixo (*least fixpoint*) de* F.

Definição 1 *Se* $(L_1, \sqsubseteq_1)$ *e* $(L_2, \sqsubseteq_2)$ *são* reticulados, *então* (α, γ) *é uma Conexão de Galois entre* L_1 *e* L_2, *se e somente se,* $\alpha \in L_1 \to L_2$ *e* $\gamma \in L_2 \to L_1$ *são funções tais que:*

$$\forall x \in L_1, y \in L_2 \cdot \alpha(x) \sqsubseteq_2 y \Leftrightarrow x \sqsubseteq_1 \gamma(y)$$

Por exemplo, voltando ao programa fatorial, o analisador de sinais baseado na abordagem "clássica" seria definido por um conjunto de equações recursivas construídas a partir do diagrama de fluxo de controle. Teríamos então, os seguintes passos para definir o analisador de sinais a partir da abordagem "clássica". Primeiramente, definiríamos a *collecting semantics* que tem como função coletar os valores que as variáveis podem assumir em um determinado ponto do programa. A *collecting semantics* é definida por um conjunto de equações recursivas construído a partir do diagrama de fluxo de controle. Cada nó do diagrama é associado, a uma (ou mais) equações. Por exemplo, no caso do programa fatorial, teríamos o seguinte conjunto de equações.

$$\begin{aligned}
Fx_1(X,Y) &= \mathbb{N} \\
Fx_2(X,Y) &= (Fx_1(X,Y) \cup Fx_4(X,Y)) \cap \{i \mid i > 1\} \\
Fx_3(X,Y) &= Fx_2(X,Y) \\
Fx_4(X,Y) &= \{x - 1 \mid x \in Fx_3(X,Y)\} \\
Fy_1(X,Y) &= \{1\} \\
Fy_2(X,Y) &= Fy_1(X,Y) \cup Fy_4(X,Y) \\
Fy_3(X,Y) &= \{x \times y \mid x \in Fx_2(X,Y) \wedge y \in Fy_2(X,Y)\} \\
Fy_4(X,Y) &= Fy_3(X,Y)
\end{aligned}$$

Em geral, não é gerada uma equação para cada ponto do programa, mas sim para cada *basic block* [1, 68], que nada mais é do que uma sequência de pontos de controle onde não há um desvio. Assim, se estivéssemos utilizando *basic blocks* as equações associadas aos pontos 2,3 e 4 seriam "fundidas".

Um conjunto de equações recursivas é resolvido (na abordagem clássica) pelo método iterativo. Inicialmente, o valor de todos os F_n são considerados iguais a $\perp$ [6]. Em seguida, recalculamos o valor de cada F_n em função dos valores das $F_n s$ consideradas no passo anterior. O valor de F_n continua a ser recalculado iterativamente, até que o valor obtido em um passo seja igual ao valor obtido no passo anterior. O método iterativo garante convergência quando todas as F_n são monótonas, e a altura do reticulado é finita. A seguir descrevemos o método iterativo para o conjunto de equações acima, onde F_n^0 representa o valor inicial de cada F_n, e as equações F_n^i descrevem o passo iterativo.

[6] Lembre que no reticulado de subconjuntos dos números inteiros, temos que $\perp = \emptyset$.

$$
\begin{aligned}
F^0x_1(X,Y) &= \bot \\
F^0x_2(X,Y) &= \bot \\
F^0x_3(X,Y) &= \bot \\
F^0x_4(X,Y) &= \bot \\
F^0y_1(X,Y) &= \bot \\
F^0y_2(X,Y) &= \bot \\
F^0y_3(X,Y) &= \bot \\
F^0y_4(X,Y) &= \bot
\end{aligned}
$$

$$
\begin{aligned}
F^ix_1(X,Y) &= \mathbb{N} \\
F^ix_2(X,Y) &= (F^{i-1}x_1(X,Y) \cup F^{i-1}x_4(X,Y)) \cap \{i \mid i > 1\} \\
F^ix_3(X,Y) &= F^{i-1}x_2(X,Y) \\
F^ix_4(X,Y) &= \{x - 1 \mid x \in F^{i-1}x_3(X,Y)\} \\
F^iy_1(X,Y) &= \{1\} \\
F^iy_2(X,Y) &= F^{i-1}y_1(X,Y) \cup F^{i-1}y_4(X,Y) \\
F^iy_3(X,Y) &= \{x \times y \mid x \in F^{i-1}x_2(X,Y) \wedge y \in F^{i-1}y_2(X,Y)\} \\
F^iy_4(X,Y) &= F^{i-1}y_3(X,Y)
\end{aligned}
$$

O próximo passo é a definição da *Conexão de Galois*, pois ao tentarmos resolver o conjunto de equações acima utilizando o método iterativo para o cálculo do menor ponto fixo, esta computação não convergirá, visto que a altura do reticulado de subconjuntos de números inteiros não é finita [7]. Por isso, devemos utilizar uma aproximação, que no nosso caso é considerar apenas os sinais das variáveis. Para isso, devemos definir uma *Conexão de Galois* entre o reticulado de subconjuntos dos números naturais e o reticulado de sinais. Podemos, então, redefinir as equações utilizando os operadores do reticulado de sinais, assim obtemos:

$$
\begin{aligned}
Fx_1(X,Y) &= \top \\
Fx_2(X,Y) &= (Fx_1(X,Y) \sqcup Fx_4(X,Y)) \sqcap pos \\
Fx_3(X,Y) &= Fx_2(X,Y) \\
Fx_4(X,Y) &= Fx_3(X,Y) \hat{-} pos \\
Fy_1(X,Y) &= pos \\
Fy_2(X,Y) &= Fy_1(X,Y) \sqcup Fy_4(X,Y) \\
Fy_3(X,Y) &= Fx_2(X,Y) \hat{\times} Fy_2(X,Y) \\
Fy_4(X,Y) &= Fy_3(X,Y)
\end{aligned}
$$

Ao calcularmos o ponto fixo deste conjunto de equações utilizando o método iterativo [1, 68], obteremos um resultado semelhante ao obtido utilizando a nossa abordagem (a Figura 2.12 contém os passos realizados na computação iterativa do ponto fixo). O número de passos necessários para calcular o ponto fixo pode ser reduzido através da utilização de algoritmos mais sofisticados que aceleram a propagação dos valores (vide Seção 10.8). Representações mais sofisticadas como *def-use chains* [68] ou *SSA* (*Static Single Assignment* [68]) também podem ser utilizadas para acelerar a convergência.

[7] Considere por exemplo a cadeia $\{0\} \subseteq \{0,1\} \subseteq \{0,1,2\} \subseteq \{0,1,2,3\} \subseteq \ldots$.

início

$$
\begin{array}{lcl}
Fx_1(X,Y) & = & \bot \\
Fx_2(X,Y) & = & \bot \\
Fx_3(X,Y) & = & \bot \\
Fx_4(X,Y) & = & \bot \\
Fy_1(X,Y) & = & \bot \\
Fy_2(X,Y) & = & \bot \\
Fy_3(X,Y) & = & \bot \\
Fy_4(X,Y) & = & \bot
\end{array}
$$

passo 1

$$
\begin{array}{lcl}
Fx_1(X,Y) & = & \top \\
Fx_2(X,Y) & = & (\bot \sqcup \bot) \sqcap pos = \bot \\
Fx_3(X,Y) & = & \bot \\
Fx_4(X,Y) & = & \bot \hat{-} pos = \bot \\
Fy_1(X,Y) & = & pos \\
Fy_2(X,Y) & = & \bot \sqcup \bot = \bot \\
Fy_3(X,Y) & = & \bot \hat{\times} \bot = \bot \\
Fy_4(X,Y) & = & \bot
\end{array}
$$

passo 2

$$
\begin{array}{lcl}
Fx_1(X,Y) & = & \top \\
Fx_2(X,Y) & = & (\top \sqcup \bot) \sqcap pos = pos \\
Fx_3(X,Y) & = & \bot \\
Fx_4(X,Y) & = & \bot \hat{-} pos = \bot \\
Fy_1(X,Y) & = & pos \\
Fy_2(X,Y) & = & pos \sqcup \bot = pos \\
Fy_3(X,Y) & = & pos \hat{\times} \bot = \bot \\
Fy_4(X,Y) & = & \bot
\end{array}
$$

passo 3

$$
\begin{array}{lcl}
Fx_1(X,Y) & = & \top \\
Fx_2(X,Y) & = & pos \\
Fx_3(X,Y) & = & pos \\
Fx_4(X,Y) & = & \bot \hat{-} pos = \bot \\
Fy_1(X,Y) & = & pos \\
Fy_2(X,Y) & = & pos \sqcup \bot = pos \\
Fy_3(X,Y) & = & pos \hat{\times} pos = pos \\
Fy_4(X,Y) & = & \bot
\end{array}
$$

passo 4

$$
\begin{array}{lcl}
Fx_1(X,Y) & = & \top \\
Fx_2(X,Y) & = & pos \\
Fx_3(X,Y) & = & pos \\
Fx_4(X,Y) & = & pos \hat{-} pos = \top \\
Fy_1(X,Y) & = & pos \\
Fy_2(X,Y) & = & pos \sqcup \bot = pos \\
Fy_3(X,Y) & = & pos \hat{\times} pos = pos \\
Fy_4(X,Y) & = & pos
\end{array}
$$

passo 5

$$
\begin{array}{lcl}
Fx_1(X,Y) & = & \top \\
Fx_2(X,Y) & = & (\top \sqcup \top) \sqcap pos = pos \\
Fx_3(X,Y) & = & pos \\
Fx_4(X,Y) & = & pos \hat{-} pos = \top \\
Fy_1(X,Y) & = & pos \\
Fy_2(X,Y) & = & pos \sqcup pos = pos \\
Fy_3(X,Y) & = & pos \hat{\times} pos = pos \\
Fy_4(X,Y) & = & pos
\end{array}
$$

Figura 2.12: Cálculo iterativo do ponto fixo

Como pudemos observar, a segurança da análise é baseada no fato das funções F serem monótonas e na definição da *Conexão de Galois*. Portanto, podemos relacionar a nossa abordagem com a abordagem "clássica", se observarmos que se $AbsVal$ for um reticulado completo, e *safe* tiver a propriedade de que c *safe* $\sqcap \{a' \mid c \text{ safe } a'\}$ então, *safe* define uma *Conexão de Galois* entre $\wp(Val)$ e $AbsVal$ ($\alpha : \wp(Val) \to AbsVal$ e $\gamma : AbsVal \to \wp(Val)$).

$$
\begin{array}{rcl}
\alpha(S) & = & \bigsqcup_{c \in S} \{\sqcap \{a \mid c \text{ safe } a\}\} \\
\gamma(a) & = & \{c \mid c \text{ safe } a\}
\end{array}
$$

Note que, $\sqcap\{a \mid c \text{ safe } a\}$ é a melhor aproximação para c, visto que para todo a' tal que c *safe* a', temos que $a' \sqsubseteq \sqcap\{a \mid c \text{ safe } a\}$. Portanto, $\bigsqcup_{c \in S}\{\sqcap\{a \mid c \text{ safe } a\}\}$ é a melhor aproximação para o conjunto S, já que este é o *join* das melhores aproximações dos elementos c contidos em S. O conjunto $\{c \mid c \text{ safe } a\}$ contém todos os valores aproximados por a.

2.6 Terminação da Análise

Verificar propriedades de programas é em geral indecidível (*teorema de Rice* - Seção 1.2) em linguagens *Turing* completas. Analisadores de propriedades podem, a princípio, entrar em *loop*. Portanto, uma questão fundamental é determinar se um analisador termina ou não, isto é, se este produz uma resposta em tempo finito ou não. Em nosso "framework" podemos garantir a terminação de um analisador usando argumentos baseados na observação do domínio semântico abstrato e suas implicações em relação a quantidade de estados abstratos que um programa possa assumir.

```
visited-states = ∅; worklist = {S_0};
while (worklist ≠ ∅)
{
  curr-state = remove-an-element(worklist);
  visited-states = visited-states ∪ {curr-state};
  for each succ-state in successors(curr-state)
  {
    connect(curr-state, succ-state);
    if (succ-state ∉ visited-states)
    { worklist = worklist ∪ {succ-state}; }
  }
}
```

Figura 2.13: Algoritmo de análise para número finito de estados

2.6.1 Número Finito de Estados Abstratos

Se o número de estados abstratos que um programa puder assumir é finito, então obviamente o analisador irá terminar. Por exemplo, o analisador de sinais da linguagem imperativa simples é um representante desta categoria, porque um programa possui um número finito de variáveis, que podem assumir uma quantidade finita de valores e este possui um número finito de pontos de controle. É prudente lembrar que o fato de o reticulado de sinais ser finito não implica em existir uma quantidade finita de estados (ex.: linguagens que possuem suporte a alocação dinâmica de memória).

Observe que ao contrário do que é normalmente encontrado na literatura de compiladores [1, 68], nenhum argumento sobre monotonicidade ou propriedades do reticulado (ou ordem parcial) de valores abstratos precisa ser utilizado.

O algoritmo de análise está descrito na Figura 2.13. A variável *visited-states* contém os estados já visitados e *worklist* contém os estados a serem visitados. O algoritmo termina quando não existem mais estados a serem visitados. A variável *curr-state* contém o estado corrente a ser processado e a função *successors* retorna os sucessores de um estado. A função *sucessors* pode ser vista como a execução de um passo computacional. A função *connect* serve para construir o mapa de estados, i.e. conecta dois nodos (estados). Um sucessor é inserido na *worklist* se este ainda não tiver sido visitado.

2.6.2 Semi-Reticulado com Altura Finita

Caso o número de estados não seja finito, temos que forçar a terminação da análise por outros meios. Podemos, por exemplo, "compactar" os *traces* divergentes. Onde o termo "compactar" significa obter uma representação finita e segura de um *trace* divergente (infinito). Uma técnica de "compactação" consiste em "fundir" todos os estados associados a um mesmo ponto de controle do programa [87]. A operação de "fusão" nada mais é do que a aplicação do operador $\sqcup$ (*join*). É claro que não podemos construir um *trace* infinito e em seguida aplicarmos esta "compactação". Por isso, devemos construir o mapa de estados ao mesmo tempo que realizamos a "compactação". Para exemplificar esta técnica, considere a linguagem imperativa simples descrita anteriormente, e o analisador "propagador de constantes". Este analisador é extremamente comum em compiladores, e serve para detectar quando em um determinado ponto do programa uma variável *sempre* assume um valor constante. O reticulado de valores associado a este ana-

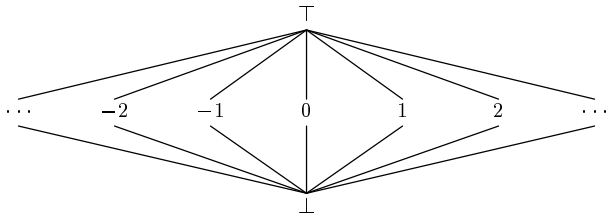

Figura 2.14: Diagrama de Hasse do reticulado de constantes

```
x := 10;
y := 1; L1:
while (x > 5)
{
  L2: x := x + y;
  L3: z := y + 1;
  L4: print(z);
}
L5:
```

Figura 2.15: Programa exemplo do "Propagador de Constantes"

lisador é descrito pelo diagrama de *hasse* contido na Figura 2.14. Como este reticulado possui uma quantidade infinita de valores [8], um programa pode possuir, a princípio, uma quantidade infinita de estados.

Considere agora, o programa descrito na Figura 2.15. Este programa possui um mapa de estados infinito que é descrito parcialmente na Figura 2.16. Para forçarmos a terminação da análise, podemos modificar o processo de geração do mapa de estados, de tal forma, que seja possível realizar a "compactação" ao mesmo tempo. Esta modificação é simples, porque precisamos apenas aplicar o operador $\sqcup$ (*join*) ao encontrarmos um estado cujo ponto de controle já foi associado a outro estado. O algoritmo contendo esta modificação é descrito na Figura 2.17. Este algoritmo é extremamente semelhante ao descrito na Figura 2.13. A principal diferença está na forma como o conjunto *worklist* é atualizado. A função *get-control-point* retorna o ponto de controle associado a um estado. A função *search-control-point* procura em um conjunto de estados, um estado que esteja associado ao ponto de controle desejado. O comando *update-state-store(same-control-point-state, join-state)* atualiza o mapa de estados, substituído *same-control-point-state* por *join-state*. No exemplo da Figura 2.15, este algoritmo obtém o mapa de estados descrito na Figura 2.18.

Este algoritmo garante convergência, porque temos uma quantidade finita de pontos de controle e durante a execução cada ponto de controle assume uma seqüência convergente de estados $(S_1, \ldots, S_n)$, onde $S_i = S_{i-1} \sqcup \Delta_i$ e conseqüentemente $S_{i-1} \sqsubseteq S_i$, i.e. estamos sempre adicionando informação aos estados associados a cada ponto de controle. Estas seqüências são convergentes, porque a altura do reticulado é finita, ou seja, o tamanho da maior seqüência estritamente crescente é finito. Por exemplo, durante a execução do algoritmo sobre o programa

[8] Na verdade, temos apenas uma quantidade finita de valores, pois, em geral, em computadores um número inteiro é representado por um quantidade fixa de *bits* (ex. 16 ou 32 bits). O reticulado, assim, seria finito permitindo, a princípio, a aplicação da técnica anterior. Entretanto, isso não acontece na prática, porque o reticulado possui uma quantidade de valores muito grande (ex. 2^{16} ou 2^{32}) o que implicaria numa quantidade de estados enorme, tornando o analisador lento e consumidor de muita memória.

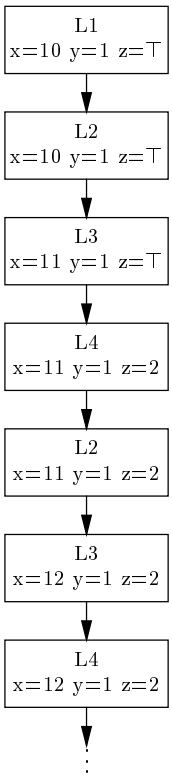

Figura 2.16: Mapa de estados do programa exemplo

da Figura 2.15, o ponto de controle *L2* assume primeiro o *store* $[x = 10, y = 1, z = \top]$, continuando a execução do algoritmo este assume o *store* $[x = \top, y = 1, z = \top]$. De uma forma "grosseira" podemos dizer que a seqüência de *stores* não pode "crescer" indefinidamente. Já que a altura do reticulado é finita, existem duas opções: convergir no "meio do caminho" ou convergir no "topo do reticulado". No caso do programa da Figura 2.15, o "topo do reticulado" é $[x = \top, y = \top, z = \top]$

Na Figura 2.19 apresentamos uma pequena variação do algoritmo da Figura 2.17, que realiza a fusão de uma forma diferente. As principais diferenças são:

- as variáveis *visited-states* e *worklist* são listas;
- *worklist* é uma lista de pares $\langle$ *state*, *path* $\rangle$. Onde *state* é o estado a ser processado e *path* é a lista de estados encontrados desde a última operação *join*;
- os estados gerados nunca são atualizados [9]. Ou seja, sempre é gerado um *novo* estado contendo a informação compactada;
- a função *cons* adiciona um elemento no início da lista;
- a função *remove-first* remove o primeiro elemento da lista;
- a função *is-in* verifica se um elemento está dentro da lista;
- a função *search-control-point* procura o elemento (i.e. estado) da lista cujo ponto de controle associado é igual ao valor especificado;

[9] Isto é, após um estado ter sido adicionado ao mapa de estados, este não é mais modificado.

```
visited-states = ∅; worklist = {S0};
while (worklist ≠ ∅)
{
  curr-state = remove-an-element(worklist);
  visited-states = visited-states ∪ {curr-state};
  for each succ-state in successors(curr-state)
  {
    if ¬(succ-state ∈ visited-states)
    {
      same-control-point-state = search-control-point(visited-states,
                                 get-control-point(succ-state));
      if (same-control-point-state == nil)
      {
        connect(curr-state, succ-state);
        worklist = cons(succ-state, worklist);
      }
      else
      {
        join-state = same-control-point-state ⊔state succ-state;
        connect(curr-state, join-state);
        update-state(same-control-point-state, join-state);
        worklist = worklist ∪ {join-state};
      }
    }
    else{ connect(curr-state, succ-state);}
  }
}
```

Figura 2.17: Algoritmo de análise usando *join*

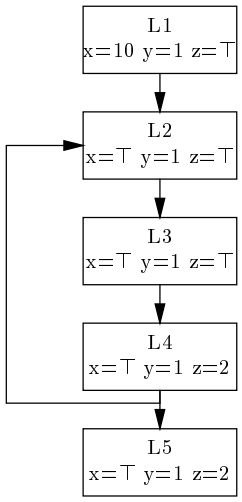

Figura 2.18: Mapa de estados do programa exemplo

```
visited-states = empty-list;
worklist = cons(⟨S_0, empty-list ⟩, empty-list);
while (worklist ≠ empty-list)
{
  ⟨ curr-state, curr-path ⟩ = remove-first(worklist);
  visited-states = cons(curr-state, visited-states);
  curr-path = cons(curr-state, curr-path);
  for each succ-state in successors(curr-state)
  {
    if ¬(is-in(succ-state,visited-states))
    {
      same-control-point-state = search-control-point(curr-path,
                                   get-control-point(succ-state));
      if (same-control-point-state = nil)
      {
        connect(curr-state, succ-state);
        worklist = cons(⟨ succ-state, curr-path ⟩, worklist);
      }
      else
      {
        join-state = same-control-point-state ⊔_state succ-state;
        connect(curr-state, join-state);
        if ¬(is-in(join-state, visited-states))
        { worklist = cons(⟨ join-state, empty-list⟩, worklist); }
      }
    }
    else
    { connect(curr-state, succ-state); }
  }
}
```

Figura 2.19: Algoritmo 2 de análise usando *join*

- toda a vez que um estado é "fundido", o campo *path* é reinicializado com a lista vazia. Por conseguinte, um estado somente será "fundido" novamente se um outro ciclo de pontos de controle for detectado.

Apesar deste algoritmo ser menos eficiente, este produz um mapa de estados mais preciso que o algoritmo anterior. Esta precisão adicional só se justifica em métodos de verificação onde reticulados mais complexos são utilizados.

2.6.2.1 Comparação com a Abordagem Tradicional

O primeiro algoritmo produz um resultado extremamente semelhante ao obtido com o método convencional (utilizando equações recursivas + cálculo do ponto fixo). O que não é totalmente extraordinário, já que "fundimos" todos os estados associados a um mesmo ponto de controle. Entretanto, o segundo algoritmo aparentemente não possui equivalentes nos métodos convencionais.

$$\alpha \in Set\,Integer \rightarrow Interval\,Integer$$
$$\alpha(S) = [min(S), max(S)]$$
$$\gamma \in Interval\,Integer \rightarrow Set\,Integer$$
$$\gamma(I) = \{e \mid e \in I\}$$

Figura 2.20: Conexão de Galois entre *Conjunto de Inteiros* e *Intervalo de Inteiros*

$$\alpha(\{1, 4, 7\}) = [1, 7]$$
$$\gamma([1, 7]) = \{1, 2, 3, 4, 5, 6, 7\}$$

Figura 2.21: Exemplo de uso da Conexão de Galois

2.6.3 Ordens parciais com Altura Infinita

No caso do reticulado ou ordem parcial de valores abstratos não possuir altura finita, o método descrito anteriormente não pode ser aplicado. Porém, este pode ser "adaptado" facilmente, se substituirmos o operador $\sqcup$ (*join*), por outro que garanta a convergência. Denominaremos este operador de $\bigtriangledown$ (*widening*). Como exemplo, assuma novamente a linguagem imperativa simples e o analisador de intervalos, que tem como objetivo inferir o intervalo de valores que uma variável pode assumir. A relação entre os reticulados de intervalos e conjuntos de números inteiros é dada pela *Conexão de Galois* descrita na Figura 2.20 (a Figura 2.21 mostra exemplos de uso das funções α e γ desta *Conexão de Galois*). Neste exemplo, o operador de *widening* pode ser definido conforme a Figura 2.22. Considere, então, o programa descrito na Figura 2.23. A partir do primeiro algoritmo descrito na seção anterior, e substituindo $\sqcup$ por $\bigtriangledown$, iremos obter o mapa de estados descrito na Figura 2.24. Com o segundo algoritmo, que é mais preciso, obteremos o mapa de estados descrito na Figura 2.25, que é mais preciso do que o mapa obtido pelo primeiro algoritmo (ex.: o segundo algoritmo conseguiu inferir que o valor da variável x é sempre maior que 12 no ponto de controle *L6*).

A definição de um operador de *widening* nem sempre é simples. Podemos definir uma categoria auxiliar de operadores que facilitem a definição de operadores de *widening*. Estes operadores auxiliares são denominados de operadores de *partition*, e os representaremos pelo símbolo $\odot$. Estes operadores possuem a propriedade de levar um elemento de um reticulado infinito em um elemento de um subconjunto finito do reticulado. O operador *widening* pode,

$$\begin{aligned}[l_1, u_1] \bigtriangledown [l_2, u_2] \quad = \quad & [\text{if } l_2 < l_1 \text{ then } -\infty \text{ else } l_1, \\ & \text{if } u_2 > u_1 \text{ then } +\infty \text{ else } u_1]\end{aligned}$$

Figura 2.22: *Widening* para *Intervalo de Inteiros*

```
INPUT x;
L1: while (x ≥ 10)
{
  L2: if (x == 10)
      { L3: y = 2; }
      else
      { L4: y = 1; }
  L5: x = x + y;
  L6:
}
L7:
```

Figura 2.23: Programa exemplo do "Analisador de Intervalos"

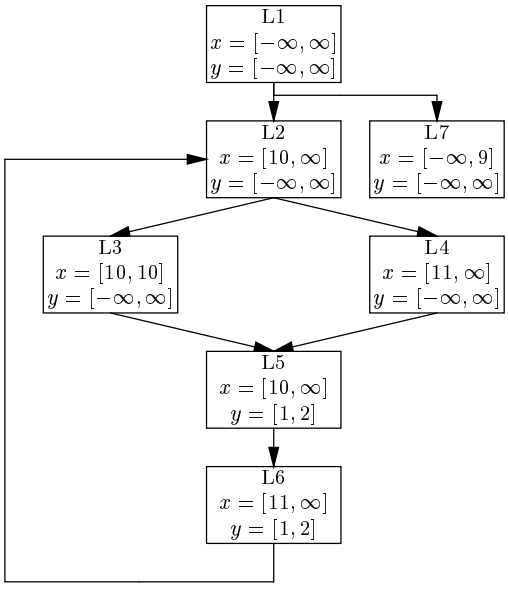

Figura 2.24: Mapa de estados do programa exemplo (algoritmo 1)

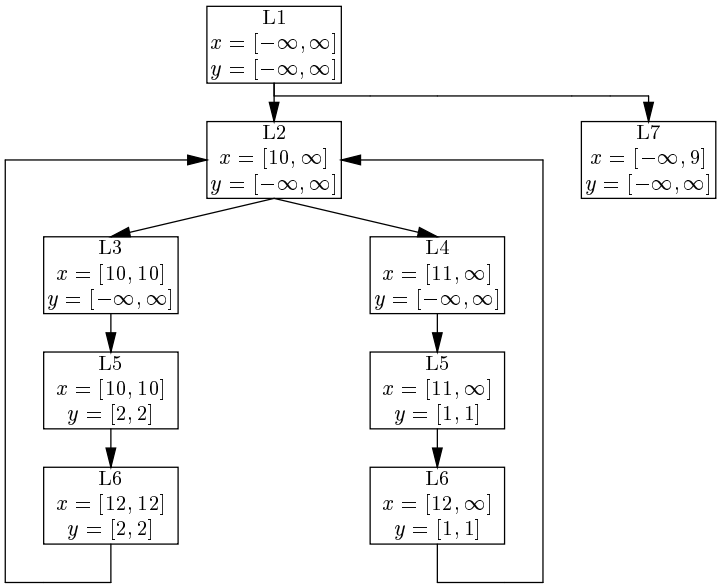

Figura 2.25: Mapa de estados do programa exemplo (algoritmo 2)

então, ser definido a partir deste operador de *partition*, como $x \bigtriangledown y \stackrel{\text{def}}{=} \odot(x \sqcup y)$. Apesar do operador *partition* possuir uma definição ingênua, mostraremos que este é muito útil na prática. Observe que a aplicação do operador *partition*, geralmente envolve a definição de um novo reticulado, que possui novos elementos que aproximam os elementos do reticulado antigo.

O operador *partition* é interessante de ser utilizado em conjunto com o algoritmo da Figura 2.19, visto que este algoritmo minimiza o uso do operador *join* ou *widening*, permitindo que representações complexas sejam utilizadas. Porém, quando um ciclo é detectado, uma aproximação de um subconjunto finito é produzida pelo operador de *partition*, garantido a convergência do processo.

2.6.3.1 Comparação com a Abordagem Tradicional

Na abordagem tradicional o operador de *widening* é utilizado durante o cálculo do ponto fixo das equações recursivas. Mais uma vez, não obtemos uma precisão equivalente a obtida com o algoritmo da Figura 2.19. Na abordagem clássica, também não está claro que o operador de *widening* é um substituto para o operador *join*. É comum encontrarmos na abordagem "clássica", o operador *join* sendo utilizado em conjunto com o operador de *widening*. Na abordagem "clássica" o operador de *widening* é definido da seguinte maneira:

Definição 2 *O operador de* widening $\bigtriangledown \in L \times L \to L$ *possui as seguintes propriedades:*

1. $\forall x, y \in L \cdot x \sqsubseteq x \bigtriangledown y$
2. $\forall x, y \in L \cdot y \sqsubseteq x \bigtriangledown y$
3. *Para toda cadeia (*chain*) não decrescente* $x^0 \sqsubseteq x^1 \sqsubseteq x^2 \sqsubseteq \ldots$*, a cadeia definida como* $y^0 = x^0$ *e* $y^{i+1} = y^i \bigtriangledown x^{i+1}$ *converge, isto é, existe n tal que* $y^{n+1} = y^n$*.*

Os elementos da cadeia y possuem a seguinte "forma": x^0, $x^0 \bigtriangledown x^1$, $x^0 \bigtriangledown x^1 \bigtriangledown x^2$, $x^0 \bigtriangledown x^1 \bigtriangledown x^2 \bigtriangledown x^3$, $\ldots$

A partir dessa definição, podemos provar o teorema a seguir, que garante que o novo cálculo iterativo do ponto fixo irá convergir.

$$[l_1, u_1] \triangle [l_2, u_2] = [\text{if } l_1 = -\infty \text{ then } l_2 \text{ else } min(l_1, l_2), \text{ if } u_1 = +\infty \text{ then } u_2 \text{ else } max(u_1, u_2)]$$

Figura 2.26: *Narrowing* para *Intervalo de Inteiros*

Teorema 3 *A cadeia(*chain*) definida como:*

- $y^0 = \bot$
- $y^{i+1} = y^i \bigtriangledown F(y^i)$

converge, e seu limite é um post fixpoint *de F.*

A abordagem clássica também possui o operador $\triangle$ (*narrowing*) [29]. A função deste operador é melhorar a precisão da análise. Na abordagem "clássica" este operador pode ser definido da seguinte forma.

Definição 3 *O operador de* narrowing $\triangle \in L \times L \to L$ *possui as seguintes propriedades:*

- $\forall x, y \in L \cdot x \sqsubseteq y \Rightarrow x \sqsubseteq x \triangle y \sqsubseteq y$
- *Para toda cadeia (*chain*) não crescente $x^0 \sqsupseteq x^1 \sqsupseteq x^2 \sqsupseteq \ldots$, a cadeia definida como $y^0 = x^0$ e $y^{i+1} = y^i \triangle x^{i+1}$ converge, isto é, existe n tal que $y^{n+1} = y^n$.*

Um teorema semelhante ao associado ao operador de *widening* pode ser demonstrado para o operador de *narrowing*.

Teorema 4 *A cadeia(*chain*) definida como:*

- $y^0 = P$, *onde P é um* post fixpoint *de F*
- $y^{i+1} = y^i \triangle F(y^i)$

converge, e seu limite é um post fixpoint *de F.*

No caso do analisador de intervalos, podemos definir o operador de *narrowing* conforme a Figura 2.26.

O operador de *narrowing* também pode ser adaptado para a nossa abordagem baseada em mapas de estados. Em suma, sempre que um *loop* é detectado, é realizada uma nova iteração usando o operador de *narrowing* para "melhorar" o resultado da análise.

2.7 Conclusão

Neste capítulo apresentamos a nossa versão para a teoria de interpretações abstratas utilizando mapas de estados e mostramos a sua relação com a abordagem "clássica". Também apresentamos diversos algoritmos que são utilizados em nossos analisadores. A nossa versão da teoria de Interpretações Abstrata torna clara a relação com a teoria de Verificação de Modelos que será apresentada no próximo capítulo.

2.7.1 Contribuições

- Relacionamos a abordagem baseada em mapas de estados (*traces*) com a abordagem "clássica".
- Reformulamos e simplificamos as condições de término de analisadores operando em conjuntos abstratos finitos.
- Apresentamos uma definição mais simples para o operador de *widening*, e mostramos como este pode ser utilizado de uma forma mais precisa.
- Definimos um algoritmo onde os operadores *join* (ou *widening*) não são aplicados desnecessariamente, o que garante uma precisão maior.
- Definimos um novo tipo de operador (*partition*) que pode ser utilizado para garantir a terminação da análise.
- Mostramos como o operador de *widening* pode ser utilizado na abordagem baseada em mapas de estados.

Capítulo 3

Verificação de Modelos

3.1 Introdução

Verificação de Modelos [19, 20] é utilizado para verificar propriedades de modelos de programa (especificações) que possuam um mapa de estados finito. Entretanto, no decorrer da tese veremos como esta técnica pode ser adaptada para mapa de estados infinitos.

Verificação de Modelos é uma abordagem de verificação fundamentalmente diferente dos métodos de prova baseados em assertivas e formalismo lógico para deduzir propriedades. Nestes métodos de prova, as regras de inferência da lógica são utilizadas para inferirmos que as assertivas implicam em propriedades de correção. Verificação de Modelos trata o mapa de estados do sistema como uma estrutura de *Kripke*, e a verificação consiste em checar se a estrutura é ou não um modelo para a propriedade de correção que desejamos verificar. Para muitas lógicas, se o mapa de estados é finito, checar se a estrutura (finita) de *Kripke* é um modelo para a fórmula é decidível. Para isso, basta usarmos algoritmos que percorram a estrutura de *Kripke*.

A grande vantagem do método de Verificação de Modelos é a produção de contra-exemplos, quando uma propriedade não é satisfeita. Um contra-exemplo consiste em um *trace* de execução que não possui a propriedade desejada. Este recurso é fundamental se tivermos uma visão pragmática, já que neste caso, estamos interessados em encontrar erros e não numa ferramenta que apenas confirme as nossas "crenças". A Figura 3.1 contém os principais componentes do processo de Verificação de Modelos.

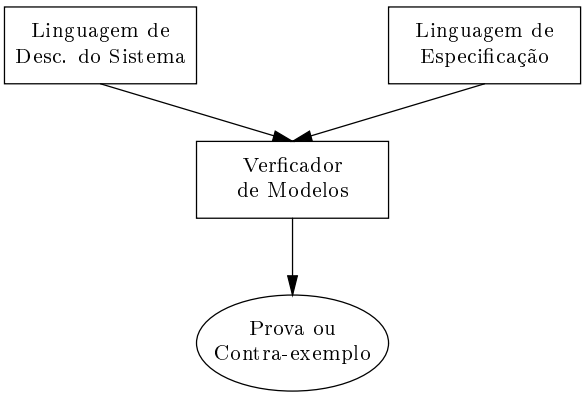

Figura 3.1: Estrutura de um Verificador de Modelos

3.2 Sistemas de Transição e Linguagens de Descrição do Sistema (*SDLs*)

Os modelos de programa utilizados em *Verificação de Modelos* são sistemas de transição finitos, que podem ser vistos como mapas de estados. Um sistema de transição consiste em um conjunto de estados e um conjunto de transições que o sistema pode efetuar. Esta definição de sistema de transição deixa claro a relação com o conceito de mapa de estados definido no capítulo anterior. Quando a modificação de estado é o resultado de um evento externo, ou uma ação feita pelo sistema, essa transição é rotulada com o evento ou ação. Os exemplos de mapas de estados apresentados no capítulo anterior não possuíam rótulos, mas isso não é uma regra.

Definição 4 *Um sistema de transição é uma quádrupla* $\mathcal{A} = \langle S, T, \alpha, \beta \rangle$ *onde*

- S *é um conjunto finito ou infinito de estados*
- T *é um conjunto finito ou infinito de transições*
- α *e* β *são dois mapeamentos de* T *em* S*, tal que para toda transição* t*,* $\alpha(t)$ *é a fonte da transição, e* $\beta(t)$ *é o destino.*

Uma transição t com fonte s e destino s' é escrita como $t : s \to s'$. Várias transições podem ter a mesma fonte e destino, i.e., o mapeamento $\langle \alpha, \beta \rangle : T \to S \times S$ não é necessariamente injetivo.

Um caminho (*path* ou *trace*) de comprimento n, $n > 0$, em um sistema de transição $\mathcal{A}$ é uma seqüencia de transições $t_1, \ldots, t_n$, tal que $\forall i : 1 \leq i < n \cdot \beta(t_i) = \alpha(t_{i+1})$. Similarmente, um caminho de tamanho infinito é uma seqüencia de transições $t_1, \ldots, t_n, \ldots$ tal que $\forall i \geq 1, \beta(t_i) = \alpha(t_{i+1})$.

Definição 5 *Um sistema de transição parametrizável é definido pela tupla* $\mathcal{A} = \langle S, T, \alpha, \beta, S_{X_1}, \ldots, S_{X_n}, T_{Y_1}, \ldots, T_{Y_m} \rangle$*. Onde* $\langle S, T, \alpha, \beta \rangle$ *é um sistema de transição, e* $(\mathcal{X}, \mathcal{Y})$ *são os parâmetros (ou propriedades), onde*

- $\mathcal{X} = \{X_1, \ldots, X_n\}$ *é um conjunto finito de nomes de parâmetros de estado*
- $\mathcal{Y} = \{Y_1, \ldots, Y_m\}$ *é um conjunto finito de nomes de parâmetros de transição*

Os subconjuntos S_X de S e T_Y de T são definidos para cada $X \in \mathcal{X}$ e $Y \in \mathcal{Y}$. Estes subconjuntos permitem que uma propriedade possa ser associada com qualquer subconjunto de estados ou transições do sistema de transição. Por exemplo, o rótulo a é a propriedade de todas as transições rotuladas com a, e o parâmetro T_a agrupa as transições que possuem esta propriedade.

Considere os dois *powersets* $A_\sigma = \wp(\mathcal{X})$ e $A_\tau = \wp(\mathcal{Y})$ e os dois mapeamentos $\mu_\sigma : S \to A_\sigma$ e $\mu_\tau : T \to A_\tau$, que são denominados de as marcações dos estados e transições, e são definidos como:

$$\begin{aligned} \mu_\sigma(s) &= \{X \mid s \in S_X\} \\ \mu_\tau(t) &= \{Y \mid t \in T_Y\} \end{aligned}$$

Estas duas funções podem também ser vistas como mapeamentos que levam estados e transições ao conjunto de propriedades que eles possuem.

Os sistemas de transição não são, em geral, especificados explicitamente. Geralmente, utilizamos uma SDL [1], com o qual descrevemos especificações/programas que irão ser transformadas em sistemas de transição. Isso não é nenhuma novidade em Interpretação Abstrata, onde os mapas de estados, que nada mais são do que sistemas de transição, são gerados a partir de programas escritos em linguagens de programação ou especificação.

3.3 Linguagens de Especificação (*SLs*)

As propriedades que desejamos verificar com um *Verificador de Modelos* são especificadas através de uma lógica. Geralmente CTL^* [20] ou mu-calculus [95] são utilizados. Do ponto de vista pragmático, essas lógicas podem ser vistas como linguagens de "consulta", que utilizamos para especificar propriedades sobre caminhos em um mapa de estados (ou sistemas de transição). Essas lógicas também são conhecidas como *transition system logics* [7].

3.3.1 Lógica Proposicional

Em lógica proposicional podemos apenas expressar propriedades dos estados e transições que não dependam da estrutura do sistema de transição. Dado um sistema de transição parametrizável

$$\mathcal{A} = \langle S, T, \alpha, \beta, S_{X_1}, \ldots, S_{X_n}, T_{Y_1}, \ldots, T_{Y_m} \rangle$$

considere a linguagem formada por:

- as constantes *true* e *false*;
- as proposições elementares P_X para todo $X \in \mathcal{X}$;
- os operadores binários $\wedge$ (conjunção) e $\vee$ (disjunção);
- o operador unário $\neg$ (negação).

As fórmulas são dadas pelas seguintes regras:

- constantes são fórmulas;
- proposições elementares são fórmulas;
- se F_1 e F_2 são fórmulas, então $F_1 \wedge F_2$ e $F_1 \vee F_2$ são fórmulas;
- se F é uma fórmula, então $\neg F$ é uma fórmula;
- nada mais é fórmula.

A relação de satisfação $\mathcal{A}, s \models F$ é definida como:

- se $F = \mathit{false}$, então $\mathcal{A}, s \not\models F$
- se $F = \mathit{true}$, então $\mathcal{A}, s \models F$
- se $F = P_X$, então $\mathcal{A}, s \models F$ se e somente se $s \in S_X$
- se $F = F_1 \wedge F_2$, então $\mathcal{A}, s \models F$ se e somente se $\mathcal{A}, s \models F_1$ e $\mathcal{A}, s \models F_2$
- se $F = F_1 \vee F_2$, então $\mathcal{A}, s \models F$ se e somente se $\mathcal{A}, s \models F_1$ ou $\mathcal{A}, s \models F_2$
- se $F = \neg F'$, então $\mathcal{A}, s \models F$ se e somente se $\mathcal{A}, s \not\models F'$

[1] Linguagem de Descrição do Sistema.

3.3.2 Lógica Temporal Linear

A lógica temporal linear expressa propriedades sobre caminhos e pode ser utilizada para estudar a evolução de um sistema ao decorrer do tempo, examinando os estados em um caminho. A lógica é chamada de linear, já que somente é considerado um único sucessor do estado corrente. A sintaxe é igual a da lógica proposicional mais os operadores:

- o operador unário N (*next*)
- o operador binário U (*until*)

A relação de satisfação $\mathcal{A}, c \models F$ é definida de forma semelhante a relação de satisfação da lógica proposicional. A diferença é que c é um caminho. A notação $c = t \cdot c'$ especifica que c é um caminho cuja a primeira transição é t e o resto do caminho é c'. Desta maneira, a relação é definida como:

- se $F = \mathit{false}$, então $\mathcal{A}, c \not\models F$
- se $F = \mathit{true}$, então $\mathcal{A}, c \models F$
- se $F = P_Y$, então $\mathcal{A}, c \models F$ se e somente se $c = t \cdot c'$ e $t \in T_Y$
- se $F = F_1 \wedge F_2$, então $\mathcal{A}, c \models F$ se e somente se $\mathcal{A}, c \models F_1$ e $\mathcal{A}, c \models F_2$
- se $F = F_1 \vee F_2$, então $\mathcal{A}, c \models F$ se e somente se $\mathcal{A}, c \models F_1$ ou $\mathcal{A}, c \models F_2$
- se $F = \neg F'$, então $\mathcal{A}, c \models F$ se e somente se $\mathcal{A}, c \not\models F'$
- se $F = N\ F'$, então $\mathcal{A}, c \models F$ se e somente se $c = t \cdot c'$ e $\mathcal{A}, c \models F'$
- se $F = F_1\ U\ F_2$, então $\mathcal{A}, c \models F$ se e somente se
  - $\mathcal{A}, c \models F_2$ ou
  - $c = t_1 \cdot t_2 \ldots t_n \cdot c'$, com $\mathcal{A}, c \models F_2$ e $\forall i \in [1, n]$, $\mathcal{A}, t_i \ldots t_n \cdot c \models F_1$

A fórmula ($\mathit{true}\ U\ F$) é normalmente abreviada como $\Diamond F$, e a fórmula $\Box F$ abrevia a fórmula $\neg\Diamond\neg F$. As fórmulas $\Diamond F$ e $\Box F$ podem ser entendidas como "eventualmente F" e "para sempre F", respectivamente.

3.3.3 *CTL* e *CTL**

CTL (*Computational Tree Logic*) foi introduzida por Clarke et al. [19, 18] para expressar propriedades de processos expressos por sistemas de transição. Essa lógica pode ser aplicada a sistemas de transição sem rótulo (ou aos mapas de estados descritos no capitulo anterior) que possuem apenas parâmetros (propriedades) de estados. Esta contém apenas fórmulas envolvendo estados, formadas a partir de:

- as constantes *true* e *false*;
- as proposições elementares P_X para todo $X \in \mathcal{X}$;
- os operadores binários $\wedge$ (conjução) e $\vee$ (disjunção);
- o operador unário $\neg$ (negação);
- os operadores unários AX e EX;

- os operadores binários $A[\cdot U \cdot]$ e $E[\cdot U \cdot]$.

Para os operadores padrão, a relação de satisfação é definida como antes. Os operadores novos são definidos como:

- $\mathcal{A}, s \models EX\ F$ se e somente se existe uma transição $s \to s'$ tal que $\mathcal{A}, s' \models F$
- $\mathcal{A}, s \models AX\ F$ se e somente se para toda transição $s \to s'$ temos que $\mathcal{A}, s' \models F$
- $\mathcal{A}, s \models E[F_1\ U\ F_2]$ se e somente se existe um caminho $c = t_1 \cdot t_2 \ldots t_n \ldots$ cuja a fonte é s, temos que
  - $\mathcal{A}, s \models F_2$ ou
  - existe um k tal que
    - $\mathcal{A}, \beta(t_k) \models F_2$ e
    - $\forall i \in [1, k] \cdot \mathcal{A}, \alpha(t_i) \models F_1$
- $\mathcal{A}, s \models A[F_1\ U\ F_2]$ se e somente se para todo caminho $c = t_1 \cdot t_2 \ldots t_n \ldots$ cuja a fonte é s, temos que
  - $\mathcal{A}, s \models F_2$ ou
  - existe um k tal que
    - $\mathcal{A}, \beta(t_k) \models F_2$ e
    - $\forall i \in [1, k] \cdot \mathcal{A}, \alpha(t_i) \models F_1$

CTL^* [21, 22, 7] é uma lógica temporal que possui fórmulas relativas a estados semelhantes a CTL e fórmulas relativas a caminhos semelhantes a lógica temporal linear. Os interessados em CTL^* devem consultar [7].

3.4 Verificando Propriedades de Sistemas de Transição

Seja $\mathcal{A}$ um sistema de transição. A fórmula F é dita válida em $\mathcal{A}$, denotada como $\mathcal{A} \models F$, se todo estado de $\mathcal{A}$ satisfaz F, i.e. $\forall s \in S \cdot \mathcal{A}, s \models F$. Uma fórmula é universalmente válida, denotada como $\models F$, se ela é válida para todo sistema de transição.

Se um processo é modelado como um sistema de transição $\mathcal{A}$, e uma propriedade é descrita por uma fórmula F (em uma lógica específica), seria interessante a computação do conjunto $\{x \mid \mathcal{A}, x \models F\}$ de objetos de $\mathcal{A}$ que satisfazem F [2]. O processo de computação é denominado de *verificação* de F. O termo $F_\mathcal{A}$ denota o conjunto $\{x \mid \mathcal{A}, x \models F\}$. Desde que a relação $\mathcal{A} \models F$ é definida por indução sobre a construção de F, o conjunto $F_\mathcal{A}$ é também definido por indução sobre a construção de F. Por exemplo, podemos escrever:

$$(F \wedge F')_\mathcal{A} = F_\mathcal{A} \cap F'_\mathcal{A}$$

considerando a lógica temporal linear, onde os conjuntos de objetos são formados por caminhos, podemos escrever:

$$\begin{aligned} (N\ F)_\mathcal{A} &= \{t \cdot c \mid c \in F_\mathcal{A}\} \\ (F\ U\ F')_\mathcal{A} &= F'_\mathcal{A} \cup \{t_1 \cdot t_2 \ldots t_n \cdot c \mid c \in F'_\mathcal{A} \wedge \forall i \in [1, n] \cdot t_i \ldots t_n \cdot c \in F_\mathcal{A}\} \end{aligned}$$

A partir de uma visão algoritmica e supondo que exista uma estrutura de dados representando o sistema de transição $\mathcal{A}$, e que existam estruturas para representar conjuntos de objetos

[2] O conjunto de elementos deve ser de um tipo bem definido, i.e. estados, transições ou caminhos.

```
visited-states = ∅; worklist = {S_0};
while (worklist ≠ ∅)
{
  curr-state = remove-an-element(worklist);
  visited-states = visited-states ∪ {curr-state};
  for each succ-state in successors(curr-state)
  {
    connect(curr-state, succ-state);
    if ¬(succ-state ∈ visited-states)
    { worklist = worklist ∪ {succ-state}; }
  }
}
```

Figura 3.2: Algoritmo de geração do sistema de transição

$X_1, \ldots, X_n$, então, para "computarmos" uma fórmula F é suficiente que para cada operador ω (da linguagem) exista um algoritmo que compute $\omega_{\mathcal{A}}(X_1, \ldots, X_n)$. Se, como é normalmente feito na prática, somente sistemas de transição finitos são verificados e as lógicas contêm apenas fórmulas relativas a estados e transições [3] [4], então é simples encontrar estruturas que implementam os estados e as transições, e a grande maioria dos algoritmos.

Se lógicas envolvendo caminhos [5] forem utilizadas, o problema das estruturas de dados torna-se um pouco mais complicado, dado que os conjuntos podem ser infinitos. Mas, é importante ressaltar que estes conjuntos infinitos possuem representações finitas, visto que estes conjuntos são regulares.

Este algoritmo ingênuo pode ser descrito em dois passos, a geração do sistema de transição, e o cálculo do conjunto de elementos associado a uma fórmula F.

3.4.1 Gerando o sistema de transição

O algoritmo de geração do sistema de transição (Figura 3.2) é semelhante ao utilizado no capítulo anterior (Figura 2.13). As principais diferenças são:

- em interpretação abstrata, os domínios concretos são substituídos por domínios abstratos (com o intuito de garantir terminação);
- alguns algoritmos (Figuras 2.17 e 2.19) de *interpretação abstrata* "fundem" (usando o operador *join* ou *widening*) os estados associados a um mesmo ponto de controle do programa.

A semelhança entre os algoritmos fornece o ponto de ligação entre as duas teorias que será explorado em nosso "framework" de analisadores.

3.4.2 Computando a fórmula F

Após a geração da estrutura de dados que representa o sistema de transição, utilizamos algoritmos que percorrem esta estrutura com o intuito de "computar" uma fórmula F. Para

[3] Como é o caso de CTL.

[4] Em um sistema de transição finitio, o conjunto de estados e transições são finitos, porém o conjunto de caminhos pode ser infinito.

[5] Como é o caso de lógica temporal linear.

```
result = ∅;
for each state s in A
    s.visited = false;
for each state s in A
    if (¬ s.visited) au(s);
```

Figura 3.3: Computando $(A[F\ U\ F'])_{\mathcal{A}}$

```
function au(s)
{
  s.visited = true;
  if (s ∈ F'_A)
    result = result ∪{s}
  else if (s ∈ F_A)
  {
    b = true;
    for each s' in successors(s)
    {
      if (¬ s'.visited) au(s');
      b = b ∧(s' ∈ result)
    }
    if (b) result = result ∪{s}
  }
}
```

Figura 3.4: Computando au(s)

"computarmos" uma fórmula F basta que, para cada operador ω exista um algoritmo que compute $\omega_{\mathcal{A}}(X_1, \ldots, X_n)$. Estes algoritmos são, em geral, simples de serem implementados, por exemplo as Figuras 3.3 e 3.4 contêm o algoritmo que computa o operador $A[\cdot U \cdot]$ de *CTL*. A Figura 3.4 contem a definição da função auxiliar *au*. Para calcularmos a fórmula $A[F\ U\ F']$, devemos resolver primeiro as fórmulas F e F', que irão gerar os conjuntos $F_{\mathcal{A}}$ e $F'_{\mathcal{A}}$. De posse destes conjuntos, devemos então aplicar o algoritmo descrito para obtermos o conjunto solução $A[F\ U\ F']_{\mathcal{A}}$.

3.5 Outras técnicas

3.5.1 *BDD* (*Binary Decision Diagram*)

Como o mapa de estados é em geral muito grande, torna-se inviável representá-lo explicitamente. As técnicas baseadas em *BDDs* [16] utilizam representações simbólicas (implícitas) do sistema de transição. Nos *Verificadores de Modelos* baseados em *BDDs*, tal como o SMV [61], o estado é modelado através de uma seqüência de *bits*. Exemplo:

- Estado = (ip_1, ip_2, x, y)
- ip_1 é o *instruction pointer* associado ao processo 1 (4 bits)

- ip_2 é o *instruction pointer* associado ao processo 2 (4 bits)
- x e y são variáveis do programa (8 bits cada uma)
- número total de bits = 24 bits
- número máximo de estados possíveis = 2^{24}!

A partir dessa representação binária, é associado a cada *bit* uma variável proposicional. Portanto, no exemplo acima teremos algo como:

$$(ip_{1_0}, \ldots, ip_{1_3}, ip_{2_0}, \ldots, ip_{2_3}, x_0, \ldots, x_7, y_0, \ldots, y_7)$$

Como cada *bit* está sendo representado por uma variável proposional, podemos então utilizar fórmulas de lógica proposicional para representar conjuntos de estados. Por exemplo, a fórmula $x_0 \wedge x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_6 \wedge x_7$ representa todos os estados cujo o valor da variável x é 255. Seguindo a mesma linha de raciocínio, a relação de transição pode ser descrita através de uma fórmula $\delta(q, q')$, que é verdade se e somente se existe uma transição de q para q'. Em nosso exemplo esta relação envolveria 48 bits.

Tendo reduzido o problema a variáveis e funções booleanas, podemos então utilizar *binary decision diagrams*(*BDDs*). *BDD* é uma estrutura de dados utilizada para representar funções booleanas. Conceitualmente, podemos construir um *BDD* para uma função booleana da seguinte forma:

- Primeiro, construa uma árvore de decisão para a função desejada, obedecendo a restrição que em qualquer caminho (*path*) da raiz até uma folha as variáveis aparecem na mesma ordem.
- Depois, aplique repetidamente as seguintes regras de redução:
  1. junte nodos duplicados (mesmo label, mesmos filhos);
  2. se ambos os filhos de um nodo apontam para o mesmo nodo, então remova o nodo, pois ele é desnecessário.

O grafo dirigido acíclico (DAG) resultante é a *BDD* para a função [6].

BDDs possuem diversas propriedades úteis. Apesar de um simples argumento quantitativo mostrar que a "maioria" das funções booleanas necessitam de grandes *BDDs* [7], muitas funções comumente usadas possuem pequenos *BDDs* (Figura 3.5). Além disso, *BDDs* são de fácil manuseio. Podemos então, computar qualquer operação binária entre duas funções representadas por *BDDs* em tempo proporcional ao produto dos tamanhos dos *BDDs*. E qualquer função pode ser avaliada em tempo linear ao número de variáveis.

A ordem em que as variáveis aparecem no *BDD* é extremamente importante. Por exemplo, suponha que desejamos construir um *BDD* para a função $(x_1 \oplus y_1) \vee (x_2 \oplus y_2)$ [8], se a ordem das variavéis for x_1, y_1, x_2, y_2, obtemos então o *BDD* descrito na Figura 3.6. Se utilizarmos a ordem x_1, x_2, y_1, y_2 obtemos um *BDD* bem maior, que está descrito na Figura 3.7. Em geral, a escolha da ordem pode transformar um *BDD* de tamanho linear em um exponencial. Como o problema de encontrar a melhor ordem é *co-NP-completo*, diversas heurísticas são utilizadas.

[6] Uma variação deste algoritmo foi implementada por nós, e utilizado para gerar automaticamente todas as figuras envolvendo *BDDs* utilizadas nesta tese.

[7] Existem 2^{2^n} funções booleanas de n variáveis. Uma esquema de representação por algum polinômio $P(n)$ pode representar no máximo $O(2^{P(n)})$ funções.

[8] O símbolo $\oplus$ representa o operador *xor*.

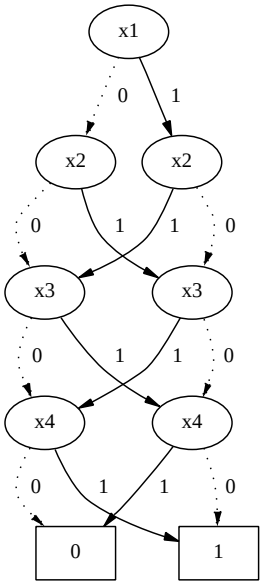

Figura 3.5: Exemplo de XOR

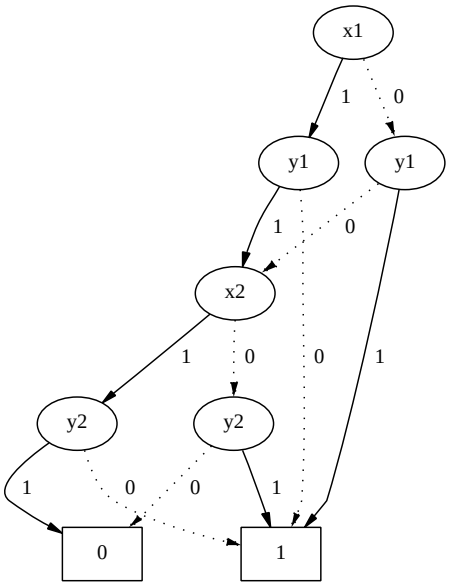

Figura 3.6: Exemplo de *BDD* com "boa ordem"

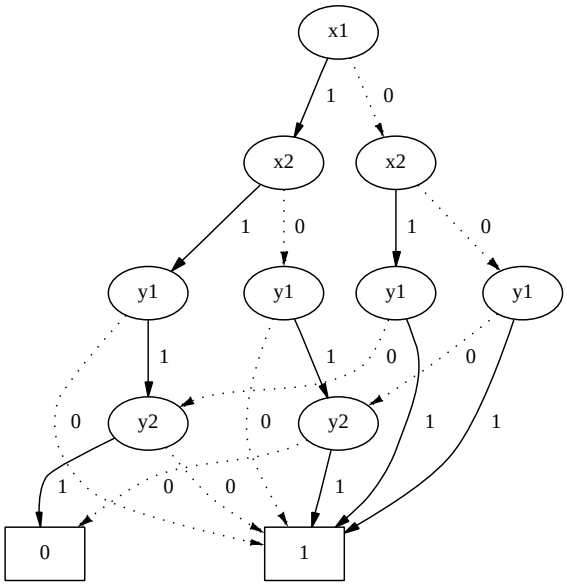

Figura 3.7: Exemplo de *BDD* com "má ordem"

O conjunto de elementos associado a uma fórmula F pode também se computado implicitamente através de *BDDs*. Os algoritmos descritos na Seção 3.4.2 podem ser substituídos por versões simbólicas. Por exemplo, considere a fórmula

$$(F \wedge F')_{\mathcal{A}} = F_{\mathcal{A}} \cap F'_{\mathcal{A}}$$

Em vez de fazermos a interseção dos elementos dos conjuntos $F_{\mathcal{A}}$ e $F'_{\mathcal{A}}$, iremos aplicar uma função que faz o "e" dos *BDDs* associados as fórmulas F e F'. As estruturas de dados para representar o sistema de transição e os conjuntos de estados, transições e caminhos são substituídas por *BDDs*.

3.5.2 Supertrace

O número de estados possíveis em um programa é muito grande, por isso, mesmo técnicas de compactação como *BDDs* não podem ser aplicadas. O algoritmo de *Supertrace* [46], introduzido pelo sistema de verificação SPIN [47], é uma variação do método simples descrito anteriormente. A principal diferença está na forma como o mapa de estados é armazenado. No método simples, e em analisadores de fluxo de dados, o sistema de transição é armazenado utilizando uma técnica denominada *hashing*. O *hashing* permite que descubramos rapidamente se um novo estado s já é ou não um membro do mapa de estados que está armazenado dentro de uma *hash table*. O método utiliza o conteúdo de s para calcular o valor de *hash* $h(s)$, que é utilizado como índice de procura na *hash table* [9]. Ao assumirmos que temos H slots em nossa *hash table*, a função $h(s)$ deve retornar um valor entre $0 \ldots (H-1)$. É fundamental observar que, podem existir dois estados diferentes s_1 e s_2 tal que $h(s_1) = h(s_2)$. No nosso caso o número de estados NE é maior que o número de slots H. Logo, a função de *hash* irá produzir o mesmo valor de *hash* para uma média de NE/H estados diferentes. Em nossa implementação, estados com o mesmo valor de *hash* serão armazenados na mesma lista. Se NE for muito maior que H a eficiência do sistema degrada rapidamente. Quanto maior for H melhor será a eficiência do sistema. Por outro lado, se H for muito grande, não sobrará muito espaço (memória) para armazenar os

[9] Existem diversas formas de implementar uma *hash table*, mas em nosso sistema ela é implementada como um array de listas.

```
bitarray = new bit[10000000]; worklist = {S_0};
while (worklist ≠ ∅)
{
  curr-state = remove-an-element(worklist);
  bitarray[ hash(curr-state) ] = true;
  for each succ-state in successors(curr-state)
  {
    if ¬(bitarray[hash(succ-state)])
    { worklist = worklist ∪ {succ-state}; }
  }
}
```

Figura 3.8: Algoritmo *Supertrace*

estados. O algoritmo de *Supertrace* resolve este problema não armazenando os estados, ou seja, ele trabalha apenas com o *hashing*. No caso do *Supertrace*, a *hashtable* é substituída por um mapa de bits (Figura 3.8), que indica se um estado já foi visitado ou não. Como conseqüência dessa abordagem, o *Supertrace* não garante cobertura total [10], se existe s_1 e s_2 tal que $h(s_1) = h(s_2)$. Entretanto, este problema é bastante minimizado, porque podemos utilizar valores de H muito maiores já que não precisamos reservar espaço para armazenar os estados. Por exemplo, se utilizarmos $100Mb$, podemos potencialmente percorrer $800M$ de estados, independentemente da quantidade de memória que um estado necessita para ser armazenado. Por deixar de realizar comparações de estado (no caso de conflito), este algoritmo é extremamente eficiente na prática. A eficiência está intrinsecamente relacionada a complexidade do algoritmo de *hash*. Uma técnica comumente utilizada para diminuir os problemas gerados por conflitos, consiste em utilizar mais de uma função de *hash*.

Como o sistema de transição não é gerado explicitamente, não podemos usar o processo em dois passos, i.e., gerar o sistema de transição e depois computar o conjunto de elementos associado a propriedade descrita pela fórmula F. Por este motivo, a verificação da fórmula F é feita em conjunto com a navegação pelos estados do sistema de transição. É obvio que, não é possível fazer esta computação para qualquer fórmula F. É importante observar que a mesma técnica pode ser utilizada na abordagem ingênua de dois passos, reduzindo o processo a um único passo.

3.5.3 Outras Variações do Método Simples

Além do *Supertrace*, existem outras variações [46] do método simples. Da mesma forma que o *Supertrace*, estas variações não garantem, em geral, cobertura total. E todas elas estão relacionadas com a obtenção dos sucessores de um estado. Podemos então, citar as seguintes variações:

- Controlada: o usuário interfere e decide que sucessor(es) do estado atual será(ão) analisado(s), isto é, serão colocados na *worklist*.
- Randomica: o(s) sucessor(es) são escolhidos de forma randomica. Esta variação produz bons resultados na prática.

[10] i.e. não garante que todos os possíveis estados de um programa serão visitados.

- Probabilística: o usuário marca o código(especificação) indicando que transições são mais "importantes". Neste caso os sucessores mais "importantes" são analisados primeiro.
- Limitada: é definido limites no tamanho das seqüências de transição.
- Mista: mistura dos métodos descritos acima.

3.5.4 Execução Livre

Para programas e especificações que possuam uma quantidade realmente enorme de estados, os métodos descritos acima não se aplicam. Nestes casos, podemos utilizar o algoritmo de execução livre [39], que navega pelos estados, sem construir o sistema de transição explicitamente, ou utilizar um *array de bits* como o método de *supertrace*. Como no método de *supertrace* as propriedades são verificadas em conjunto com a navegação pelo sistema de transição. Entretanto, o conjunto de fórmulas que pode ser utilizado é ainda mais limitado. Por esses motivos, outras técnicas de verificação de propriedades são utilizadas. Por exemplo, *deadlocks* são verificados usando uma abordagem baseada em "marcações". Nesta abordagem, o usuário decora o programa com "marcações" que indicam progresso. Durante o processo de verificação, se uma dessas marcas não for alcançada em um número x de passos, o analisador gera um "erro" (*deadlock*). É claro que, nem todos os *deadlock* detectados por essa técnica são realmente *deadlocks* (ex.: o número de passos x é muito pequeno). Em geral esse tipo de processo de verificação não termina, ou seja, ele fica rodando por tempo indeterminado a procura de "erros". Apesar de ter uma base teória muito simples, este método é bastante efetivo na prática, e se encaixa bem na visão pragmática adotada nesta tese.

3.5.5 *Partial Order Methods*

O principal limite de técnicas como *Verificação de Modelos* é a explosão do número de estados. Este problema está relacionado a diversos fatores, mas a modelagem da concorrência através do *interleaving* é o principal. Porém, explorar *todos* os possíveis *interleavings* não é, a priori, necessário para a verificação. Existe uma coleção de métodos de verificação denominada *partial-order methods* [38], que evita a exploração de todos os possíveis *interleavings*.

A intuição por trás destes métodos, é que uma execução concorrente define na verdade uma ordem parcial sobre os eventos concorrentes, e que a ordem de vários desses eventos é irrelevante, visto que estes são *não interferentes*. Do ponto de vista prático, estes métodos evitam que alguns caminhos sejam gerados/explorados, porque eles são equivalentes a outros caminhos. Por exemplo, considere o programa descrito na Figura 3.9, este programa contém dois processos que irão executar em paralelo. Observe que a variável *l1* do processo *A* e *l2* do processo *B* são locais, logo modificações nessas variáveis não interferem na execução do outro processo. Mas, a variável x é global, e uma modificação no valor dessa variável afeta todos os processos que a acessem. A Figura 3.10 contém o sistema de transição associado ao programa, onde foram ignorados os caminhos irrelevantes.

3.6 Conclusão

Neste capítulo deixamos claro a relação entre *Interpretação Abstrata* e *Verificação de Modelos*. Acreditamos que estas duas técnicas são complementares. A de Verificação de Modelos contribui com linguagens de consulta (as lógicas modais), métodos para lidar com especificações/programas concorrentes e técnicas de verificação de propriedades. Já a de Interpretação Abstrata contribui com aproximações e abstrações que podem ser utilizadas para lidar com o

A1: l1 := 0;	B1: l2 := 1;
A2: l1 := l1 + 1;	B2: l2 := l2 + 2;
A3: x := l1;	B3: x := l2;
A4:	B4:

Figura 3.9: Programa Exemplo

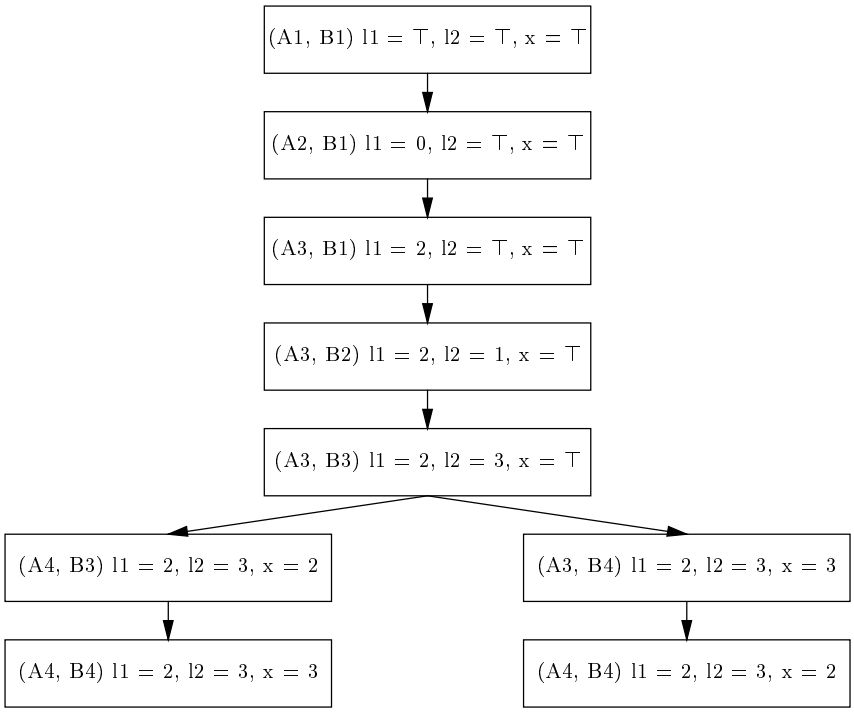

Figura 3.10: Sistema de transição usando *partial-order methods*

problema da explosão do número de estados. As aproximações podem ser utilizadas inclusive para analisarmos programas que possuam mapas de estado (sistemas de transição) infinitos.

Capítulo 4

SOS - Structural Operational Semantics

4.1 Introdução

SOS (*Structural Operational Semantics*) [78] é um formalismo desenvolvido por *Gordon Plotkin* para especificar a semântica formal de linguagens de programação. *SOS* representa uma computação através de um sistema dedutivo, transformando uma máquina abstrata em um sistema lógico dedutivo. Desde que as descrições semânticas estão codificadas em lógica, portanto, *provas* sobre propriedades de programas podem ser derivadas diretamente da definição dos construtores da linguagem, usando indução estrutural. Uma variação bem conhecida de *SOS* é *Natural Semantics* [55]. *SOS* também é denominada de *Small Step Operational Semantics* e *Natural Semantics* de *Big Step Operational Semantics*.

Em *SOS* as definições são descritas através de regras de inferência compostas de conclusões que são conseqüências de um conjunto de premissas. Geralmente as regras de inferência são descritas através de uma linha horizontal, onde as premissas e/ou condições ficam na parte superior, e a conclusão na parte inferior da linha. Este método de apresentar regras de inferência possui suas bases fundamentadas na *dedução natural* encontrada na lógica.

$$\frac{premissa_1 \; premissa_2 \; \ldots \; premissa_n}{conclusão}$$

As regras descrevem as possíveis transições (*small steps*) de uma computação. Cada *"prova"* dentro do sistema dedutivo definido descreve uma transição de estado. O estado é definido por uma *configuração*, que é uma estrutura formada por termos da linguagem e outros elementos que sejam relevantes. Por exemplo, em uma linguagem imperativa, um destes elementos é denominado *store*, e é utilizado para modelar a memória do computador.

Em *SOS* uma transição leva uma configuração em uma nova configuração, e uma computação é representada por uma seqüência de transições (*small steps*). Já em *Natural Semantics* uma transição leva uma configuração em um valor, e a computação é representada por uma única transição (*big step*). Por exemplo, no caso da linguagem imperativa simples, teríamos que:

- *SOS*: (*term*, *store*) $\rightarrow$ (*term*, *store*)
- *Natural Semantics*: (*term*, *store*) $\rightarrow$ (*store*)

Uma especificação em *SOS* (ou em *Natural Semantics*) pode também ser vista como um programa (interpretador) escrito em uma linguagem baseada em lógica (como *Prolog*). Apesar de não ser uma forma "elegante" de ver uma descrição *SOS*, esta é bastante útil e será usada freqüentemente em nosso trabalho.

$$\frac{\langle Be, Sto\rangle \rightarrow \langle Be', Sto\rangle}{\langle if\ \ Be\ \ then\ \ S, Sto\rangle \rightarrow \langle if\ \ Be'\ \ then\ \ S, Sto\rangle} \tag{4.1}$$

$$\langle if\ \ true\ \ then\ \ S, Sto\rangle \rightarrow \langle S, Sto\rangle \tag{4.2}$$

$$\langle if\ \ false\ then\ S, Sto\rangle \rightarrow \langle skip, Sto\rangle \tag{4.3}$$

$$\frac{\langle S_1, Sto\rangle \rightarrow \langle S_1', Sto'\rangle}{\langle S_1; S_2, Sto\rangle \rightarrow \langle S_1'; S_2, Sto'\rangle} \tag{4.4}$$

$$\langle skip; S, Sto\rangle \rightarrow \langle S, Sto\rangle \tag{4.5}$$

Figura 4.1: Exemplo de *SOS*

$$\langle S_1, Sto_1\rangle \twoheadrightarrow \langle S_1, Sto_1\rangle$$

$$\frac{\langle S_1, Sto_1\rangle \twoheadrightarrow \langle S_2, Sto_2\rangle, \langle S_2, Sto_2\rangle \rightarrow \langle S_3, Sto_3\rangle}{\langle S_1, Sto_1\rangle \twoheadrightarrow \langle S_3, Sto_3\rangle}$$

Figura 4.2: Fecho transitivo e reflexivo da relação $\rightarrow$

4.2 Exemplo

A Figura 4.1 contém o fragmento da semântica de uma linguagem imperativa simples. Neste exemplo, as regras 4.1, 4.2 e 4.3 descrevem o comportamento do comando *if*. A partir destas regras, podemos concluir que a avaliação da condição do comando *if* não modifica o conteúdo da memória, e que o corpo do *if* é executado apenas se a condição for avaliada como *true*.

Também é possível derivar novas regras de inferência. Por exemplo, podemos derivar a regra 4.6 utilizando o fecho transitivo e reflexivo $\twoheadrightarrow$ de $\rightarrow$, definido na Figura 4.2 [1], e as regras 4.4 e 4.5. Através de argumentos indutivos também é possível mostrar que a regra 4.7 é válida.

[1] Também podemos ver a relação $\twoheadrightarrow$ como a representação de zero ou mais "passos".

$$\frac{\langle S_1, Sto\rangle \rightarrow \langle skip, Sto'\rangle}{\langle S_1; S_2, Sto\rangle \twoheadrightarrow \langle S_2, Sto'\rangle} \tag{4.6}$$

$$\frac{\langle S_1, Sto\rangle \twoheadrightarrow \langle skip, Sto'\rangle}{\langle S_1; S_2, Sto\rangle \twoheadrightarrow \langle S_2, Sto'\rangle} \tag{4.7}$$

Figura 4.3: Regras derivadas

4.3 *SOS* Modular

Apesar de ser simples [2] definir a semântica de uma linguagem utilizando *SOS*, esta descrição não é modular. Isto é, modificações na semântica da linguagem invariavelmente implicam em mudanças globais na descrição *SOS*. Outro problema, decorrente da falta de modularidade, é a dificuldade em tratar características ortogonais da linguagem de forma independente e modular. No exemplo da Figura 4.1, se adicionarmos o conceito de ambiente de variáveis, todas as regras terão que ser modificadas, porque a estrutura da configuração será modificada de $\langle term, store\rangle$ para $\langle term, store, environment\rangle$. Mesmo as regras (ex.: regra 4.4) que não possuem nenhuma relação com o ambiente de variáveis terão que ser modificadas. Esta falta de modularidade é extremamente problemática, visto que implica em modificações globais toda a vez que a estrutura da configuração for modificada. Como a estrutura das regras foi modificada, *todas* as derivações de novas regras (ex.: regras 4.6 e 4.7) tornam-se inválidas.

Este problema é semelhante ao encontrado em semântica denotacional [96, 85]. Assim, a solução proposta para modularizar descrições *SOS* [67] foi inspirada na solução utilizada para modularizar descrições denotacionais [65, 100]. Em ambos os casos podemos entender a solução de duas formas. A primeira é do ponto de vista teórico, que recorre a teoria de categorias para motivar e justificar a construção. A outra é do ponto de vista prático, e neste caso as soluções não passam de "truques" de programação usados para modularizar o programa. E de certa forma, estes "truques" não são diferentes dos usados tradicionalmente na área de modularização de programas, visto que estes podem ser vistos como uma técnica de encapsulamento de dados.

A idéia central de *SOS* modular [67] é restringir as configurações a serem compostas apenas de termos e valores, e obrigar a relação de transição a ser ternária ($\gamma \xrightarrow{\alpha} \gamma'$). Qualquer componente adicional (tal como *store* e ambiente de variáveis) deve ser incorporado aos *labels* das transições.

Encapsulando a estrutura dos *labels*, e utilizando operações para acessar (i.e. *getters* e *setters*) e modificar componentes dos *labels* torna as regras que definem as relações de transição extremamente modulares, permitindo que estas sejam reutilizadas quando estivermos estendendo a linguagem.

A transição $\gamma \xrightarrow{\alpha} \gamma'$ representa um passo computacional. Desde que γ e γ' estão restritos a serem termos ou valores, o *label* α deve determinar o estado da informação antes e depois do passo computacional. Por exemplo, suponha que tenhamos a seguinte estrutura não modular de transição $\langle term_1, store_1\rangle \rightarrow \langle term_2, store_2\rangle$. Para colocarmos esta estrutura no formato $\gamma \xrightarrow{\alpha} \gamma'$, simplesmente definimos a estrutura do label como $\langle store_1, store_2\rangle$. Logo, teremos uma estrutura de transição $term_1 \xrightarrow{\alpha} term_2$, onde $\alpha = \langle store_1, store_2\rangle$. O próximo passo no processo de modularização, consiste em encapsularmos a estrutura do *label* e fornecermos operações [3] para acessarmos a estrutura. Neste caso, podemos definir dois operadores, um para modificar o *store* e o outro para pegar o valor de uma célula do *store*.

A modularidade foi alcançada simplesmente pelo uso de estruturas de dados e mecanismos de encapsulamento. Dentro desse ponto de vista, o *label* não passa de um tipo abstrato de dados que possui um conjunto de operações. Esta solução é extremamente semelhante a utilizada em semântica denotacional, onde *monads* fazem o papel do tipo abstrato de dados que encapsula a estrutura do domínios semânticos e possui um conjunto de operações para acessar essa estrutura. Nesta idéia de conjunto de operações é interessante a identificação de operações universais, que todo o *label* deveria possuir. Estas operações permitiriam uma caracterização dos *labels* como estrutura de dados. Através do exemplo da Figura 4.1 e a definição do fecho transitivo na Figura 4.2, fica claro que os conceitos de identidade e composição são importantes. Tal fato

[2] A palavra "simples" significa que consideramos mais fácil realizar um especificação em *SOS* do que implementar um interpretador ou compilador em um linguagens como *C*, *Pascal* ou *Java*.

[3] Usando a terminologia de orientação a objetos, podemos ver estas operações como os métodos de uma classe.

$$id(\alpha) \stackrel{\text{def}}{=} (\alpha = \langle store, store \rangle)$$

Figura 4.4: Definição do predicado *id*

$$compose(\alpha_1, \alpha_2, \alpha) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} \alpha_1 & = & \langle store_1, store_2 \rangle \wedge \\ \alpha_2 & = & \langle store_2, store_3 \rangle \wedge \\ \alpha & = & \langle store_1, store_3 \rangle \end{array} \right.$$

Figura 4.5: Definição do predicado *compose*

não é de todo estranho, já que em semântica denotacional, todo *monad* possui pelo menos os operadores *unit* e *bind* que capturam os conceitos de identidade e composição, respectivamente.

As operações sobre um label podem ser vistas como predicados que representam a propriedade desejada. Por exemplo, o operador de identidade pode ser visto como um predicado de um argumento que assegura que um *label* é a identidade. No caso da estrutura do *label* ser $\langle store_1, store_2 \rangle$, o predicado identidade pode ser definido como na Figura 4.4. Em outras palavras, o valor *inicial do store* é igual ao valor *final do store*. O operador de composição pode ser visto como um predicado de três argumentos, que assegura que o terceiro argumento é a composição dos dois primeiros. No caso da estrutura do label ser $\langle store_1, store_2 \rangle$, o predicado de composição pode ser definido como na Figura 4.5. Dois *labels* podem ser compostos, se e somente se, o valor *final do store* do primeiro *label* for igual ao valor *inicial do store* do segundo *label*.

Como o predicado de identidade (*id*) é muito utilizado, podemos então utilizar um *açúcar sintático* para simplificar a descrições *SOS*. A Figura 4.6 contém a definição desse *açúcar sintático*.

Da mesma forma que os operadores *unit* e *bind* de um *Monad* devem satisfazer a um conjunto de leis (ou propriedades), os predicados *id* e *compose* também possuem um conjunto de regras. O par de operadores *id* e *compose* é dito válido, se e somente se, a lei (regra) descrita na Figura 4.7 for válida.

De posse da definição dos predicados *id* e *compose*, podemos construir uma definição genérica para o fecho transitivo e reflexivo da relação $\rightarrow$ (Figura 4.8). Esta definição, nada mais é do que uma generalização da definição da Figura 4.2.

Nem sempre os *labels* são formados por pares de estado representando os valores antes e depois da transição. Por exemplo, os sinais de sincronização estudados em muitas algebras de processos [8] precisam somente de um estado semântico.

$$S_1 \stackrel{i}{\rightarrow} S_2 \stackrel{\text{def}}{=} \qquad \frac{id(\alpha)}{S_1 \stackrel{\alpha}{\rightarrow} S_2}$$

Figura 4.6: *Açúcar sintático* para o predicado *id*

$$\forall \alpha_1, \alpha_2 \cdot id(\alpha_1) \Rightarrow compose(\alpha_1, \alpha_2, \alpha_2) \wedge compose(\alpha_2, \alpha_1, \alpha_2)$$

Figura 4.7: Lei dos operadores *id* e *compose*

$$S_1 \stackrel{i}{\twoheadrightarrow} S_1$$

$$\frac{S_1 \stackrel{\alpha_1}{\twoheadrightarrow} S_2,\ S_2 \stackrel{\alpha_2}{\rightarrow} S_3,\ compose(\alpha_1, \alpha_2, \alpha)}{S_1 \stackrel{\alpha}{\twoheadrightarrow} S_3}$$

Figura 4.8: Fecho transitivo e reflexivo da relação $\rightarrow$

4.4 Exemplo de *SOS* Modular

A semântica descrita na Figura 4.1 pode ser reconstruída de forma modular utilizando *labels*. Nesta reformulação, utilizaremos a definição dos predicados *id* e *compose* descritos nas Figuras 4.4 e 4.5. Para acessar o conteúdo do *store* adicionaremos os predicados (*set-store* e *get-store*) descritos na Figura 4.9. Na definição destes predicados estamos assumindo que o *store* seja um *mapping* que possui operações de atualização e acesso.

A Figura 4.10 contém a definição de modular do exemplo descrito na Figura 4.1. Para evitarmos que a definição do comando *skip* esteja acoplada a definição do comando de composição seqüencial *(;)*, adicionamos um valor denominado *completed*, que representa uma computação que terminou com sucesso. A definição do comando de atribuição *(:=)* foi adicionada para exemplificarmos o uso do predicado *set-store*. Se decidirmos modificar a semântica da linguagem adicionando por exemplo um ambiente de variáveis, precisaríamos então modificar apenas algumas regras. Por exemplo, a regra que define o comando de atribuição teria que ser modificada, porque o *store* não é mais um *mapping* de identificadores em valores, e sim um *mapping* de localizações de memória em valores, e a localização de memória associada a um identificador está armazenada no ambiente de variáveis.

Além de diminuir o impacto de modificações semânticas, *SOS* modular também evita que as derivações de novas regras sejam perdidas. Por exemplo, as derivações das regras 4.8 e 4.9 não precisariam ser refeitas, se por exemplo fosse adicionado na linguagem o conceito de ambiente de variáveis. Algumas derivações podem inclusive serem consideradas globais, i.e., elas independem da estrutura do *label*. Por exemplo, a derivação descrita na Figura 4.12 é válida para qualquer *label*, já que esta não faz referência a estrutura do *label*, nem a predicado algum diferente de *id* e *compose*.

$$\begin{aligned} set\text{-}store(\alpha, Id, Val) &\stackrel{\text{def}}{=} \alpha = \langle S_1, S_2 \rangle \wedge\ S_2 = S_1[Id \leftarrow Val] \\ get\text{-}store(\alpha, Id, Val) &\stackrel{\text{def}}{=} \alpha = \langle S_1, S_2 \rangle \wedge\ Val = S_1(Id) \end{aligned}$$

Figura 4.9: Definição das operações do *label*

$$skip \xrightarrow{i} completed \qquad \frac{set\text{-}store(\alpha, Id, Val)}{Id := Val \xrightarrow{\alpha} completed}$$

$$\frac{Be \xrightarrow{\alpha} Be'}{if\ Be\ then\ S_1\ else\ S_2 \xrightarrow{\alpha} if\ Be'\ then\ S_1\ else\ S_2}$$

$$if\ true\ then\ S_1\ else\ S_2 \xrightarrow{i} S_1$$

$$if\ false\ then\ S_1\ else\ S_2 \xrightarrow{i} S_2$$

$$while\ E\ do\ S \xrightarrow{i} if\ E\ then\ S; while\ E\ do\ S\ else\ skip$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1; S_2 \xrightarrow{\alpha} S_1'; S_2} \qquad completed\ ; S_2 \xrightarrow{i} S_2$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1 \parallel S_2 \xrightarrow{\alpha} S_1' \parallel S_2} \qquad \frac{S_2 \xrightarrow{\alpha} S_2'}{S_1 \parallel S_2 \xrightarrow{\alpha} S_1' \parallel S_2}$$

$$completed \parallel completed \xrightarrow{i} completed$$

Figura 4.10: Exemplo de SOS modular

$$\frac{S_1 \xrightarrow{\alpha} skip}{S_1; S_2 \xrightarrow{\alpha\!\!\alpha} S_2} \tag{4.8}$$

$$\frac{S_1 \xrightarrow{\alpha\!\!\alpha} skip}{S_1; S_2 \xrightarrow{\alpha\!\!\alpha} S_2} \tag{4.9}$$

Figura 4.11: Regras derivadas

$$\frac{\dfrac{S_1 \xrightarrow{\alpha} skip}{S_1; S_2 \xrightarrow{\alpha} skip; S_2} \qquad \dfrac{skip \xrightarrow{i} completed}{skip; S_2 \xrightarrow{i} completed; S_2}}{S_1; S_2 \xrightarrow{\alpha\!\!\alpha} completed; S_2}$$

$$\frac{S_1; S_2 \xrightarrow{\alpha\!\!\alpha} completed; S_2 \quad completed; S_2 \xrightarrow{i} S_2}{S_1; S_2 \xrightarrow{\alpha\!\!\alpha} S_2}$$

Figura 4.12: Derivação da regra 4.8

As regras de *SOS* modular também podem ser vistas como entidades que requisitam recursos do *label*. Por exemplo, a regra que define o comando de composição seqüencial *(;)* não requisita nenhuma estrutura do *label*, utilizando apenas o predicado *id*. Já a regra que define o comando de atribuição *(:=)* requisita que o *label* possua o predicado *set-store*. De forma equivalente, as derivações também requisitam recursos. A derivação da Figura 4.12, por exemplo, não requisita estrutura alguma, esta utiliza apenas a lei dos operadores *id* e *compose* (Figura 4.7). Já outras derivações podem requisitar que determinadas propriedades sejam válidas. Uma derivação pode, por exemplo, requisitar que a seguinte propriedade seja válida

$$\forall \alpha_1, \alpha_2, i, v \cdot \mathit{composable}(\alpha_1, \alpha_2) \wedge \mathit{set\text{-}store}(\alpha_1, i, v) \Rightarrow \mathit{get\text{-}store}(\alpha_2, i, v)$$

onde *composable* é definido como

$$\mathit{composable}(\alpha_1, \alpha_2) \stackrel{\text{def}}{=} \exists \alpha \cdot \mathit{compose}(\alpha_1, \alpha_2, \alpha)$$

O que seria, mais uma vez, uma forma de desacoplar a estrutura dos *labels* das derivações de novas regras. Somente as propriedades dos predicados (operadores) do *label* são utilizadas nas derivações de novas regras e não a estrutura do *label* e/ou a definição dos predicados.

4.5 Transições Modulares

Aparentemente, uma quantidade fixa de "construções" é suficiente para modelarmos qualquer linguagem de programação. Esta situação é semelhante a escolha de domínios para representar computações em semântica denotacional, onde é utilizado métodos "fixos" para modelar conceitos como *store*, ambiente de variáveis, continuações, *power domains* e *resumptions*.

Stores Podemos obter *stores* adicionando-os nos *labels*. Estes devem ser adicionados aos pares representando, desta forma, o valor do *store* antes e depois da transição. A composição de *labels* somente é permitida conforme descrito na Figura 4.5. Devem estar disponíveis predicados para acesso, modificação e alocação.

Ambiente de Variáveis O suporte a ambientes de variáveis é provido de forma semelhante ao *store*. A diferença está na natureza de informação armazenada e nos predicados disponíveis. Devem existir predicados para estender e acessar os *bindings* de variável.

Exceções Um *flag* é adicionado ao *label* para distinguir o estado normal dos estados contendo exceções. Devem estar disponíveis predicados para modificar o conteúdo do *flag*.

Entrada e Saída Para I/O síncrono, nenhum componente precisa ser adicionado ao estado, somente ao *label*. No caso de I/O assíncrono, devemos adicionar componentes representando *buffers*, e estes componentes se comportam como estados, i.e. de forma semelhante ao *store* e ambiente de variáveis.

Nenhum suporte a não determinismo se faz necessário, ao contrário do que acontece em semântica denotacional. Entretanto, não é simples modelar continuações em *SOS*. Uma generalização do conceito de árvore de sintaxe abstrata é necessário [31].

4.6 Implementando *SOS*

Descrições *SOS* podem ser facilmente mapeadas para *Prolog* [94]. A sintaxe abstrata da linguagem pode ser modelada usando funtores. Por exemplo, o programa:

```
x := 1; skip; x := x + 1
```

é mapeado no termo *Prolog*

```
seq(assign(var("x"), num(1)),
    seq(skip, assign(var("x"),add(var("x"),num(1)))))
```

Para modelarmos as transições de *SOS*, podemos utilizar um predicado *Prolog* denominado *transition*. Este predicado possui três argumentos, representando o termo antes e depois da transição, e o label associado. Por exemplo, a transição $S_1 \xrightarrow{\alpha} S_1'$ é mapeada para:

```
transition(S1, L, S1new)
```

As regras de *SOS* são mapeadas diretamente em cláusulas de Horn. Por exemplo, a regra:

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S1; S2 \xrightarrow{\alpha} S_1'; S2}$$

é mapeada na seguinte cláusula de Horn:

```
transition(seq(S1,S2), L, seq(S1new,S2)) :-
     transition(S1, L, S1new).
```

O último detalhe diz respeito ao *label* identidade. A Figura 4.10 contém diversas regras que utilizam o *açúcar sintático* para o predicado de identidade (representado como $\xrightarrow{i}$). Na nossa implementação em *Prolog*, utilizaremos o predicado *id* explicitamente. A regra:

$$\textit{while } E \textit{ do } S \xrightarrow{i} \textit{if } E \textit{ then } S; \textit{while } E \textit{ do } S \textit{ else skip}$$

é representada pela seguinte cláusula de Horn:

```
transition(while(E,S),LABEL,
           ifthenelse(E,seq(S,while(E,S)),skip)) :- id(LABEL).
```

A Figura 4.13 contém o programa Prolog equivalente a descrição *SOS* contida na Figura 4.10.

4.7 Outras formas de obtermos *SOS* modular

A utilização de *labels* não é a única forma de obtermos modularidade [14]. Também podemos obter modularidade encapsulando a noção de estado e definindo a configuração como sendo formada de $\langle term, state \rangle$. Da mesma forma que existem operadores (predicados) para acessarem a estrutura de um *label*, podemos definir predicados para acessar a estrutura do *estado*. Porém, não seria necessário termos os predicados *id* e *compose*, como pode ser observado nas Figuras 4.14 e 4.15. Apesar desse exemplo parecer em muito com o exemplo da Figura 4.1, a principal diferença está no fato de que *state* não está necessariamente apenas representando o *store*, mas qualquer componente adicional, tal como o ambiente de variáveis. Isto é possível devido a estrutura do estado não ser acessada diretamente.

Nos casos onde a informação não faça parte do estado, esta deve fazer parte do *label*, tal como sinais de sincronização ou exceções.

```
transition(skip,LABEL,completed) :- id(LABEL).
transition(assign(ID,E),LABEL,completed) :- set(ID,E,LABEL).
transition(ifthenelse(E,S1,_),LABEL,S1) :- id(LABEL),
          eval(E, LABEL, V), V = true.
transition(ifthenelse(E,_,S2),LABEL,S2) :- id(LABEL),
           eval(E, LABEL, V), V = false.
transition(while(E,S),LABEL,
   ifthenelse(E,seq(S,while(E,S)),skip)) :- id(LABEL).
transition(seq(completed,S2), LABEL, S2) :- id(LABEL).
transition(seq(S1,S2),LABEL,seq(S1new,S2)) :-
           transition(S1,LABEL,S1new).
transition(par(completed,completed), LABEL, completed) :-
           id(LABEL).
transition(par(S1,S2),LABEL,par(S1new,S2)) :-
           transition(S1,LABEL,S1new).
transition(par(S1,S2),LABEL,par(S1,S2new)) :-
           transition(S2,LABEL,S2new).
```

Figura 4.13: Programa Prolog associado a descrição *SOS*

$$\langle S_1, State_1\rangle \rightarrow \langle S_1, State_1\rangle$$

$$\frac{\langle S_1, State_1\rangle \twoheadrightarrow \langle S_2, State_2\rangle,\ \langle S_2, State_2\rangle \rightarrow \langle S_3, State_3\rangle}{\langle S_1, State_1\rangle \twoheadrightarrow \langle S_3, State_3\rangle}$$

Figura 4.14: Fecho transitivo e reflexivo da relação $\rightarrow$

$$\langle skip, State\rangle \rightarrow \langle completed, State\rangle$$

$$\frac{set\text{-}store(State_1, Id, Val, State_2)}{\langle Id := Val, State_1\rangle \rightarrow \langle completed, State_2\rangle}$$

$$\frac{\langle S_1, State_1\rangle \rightarrow \langle S_1', State_1'\rangle}{\langle S_1; S_2, State_1\rangle \rightarrow \langle S_1'; S_2, State_1'\rangle}$$

Figura 4.15: Exemplo de *SOS* modular

4.8 Por que não usar Semântica Denotacional?

Em semântica denotacional, programas escritos em uma linguagem imperativa simples são descritos por uma função de *Store* em *Store*. Essa modelagem é constantemente referenciada como um dos pontos "fortes" de semântica denotacional e como uma das grandes vantagens sobre semântica operacional, porque a noção de passo (*step*) computacional foi abstraída e somente a "essência" do programa é mantida. Esta peculiaridade é extremamente útil quando estamos justificando transformações (otimizações) em programas, fornecendo uma forma extensional de equivalência. Os dois trechos de programa a seguir, por exemplo, são equivalentes segundo a semântica denotacional mencionada, mas ao mesmo tempo eles são diferentes se considerarmos questões relativas a eficiência.

$$D[\![x := 1; y := y + 1;]\!] = D[\![x := 1; y := 2;]\!]$$

Por outro lado, podemos ver essa equivalência, também, como uma "perda" de informação dos passos computacionais envolvidos. Outro problema diz respeito as computações potencialmente infinitas que não possuem um estado "final", como as encontradas em sistemas reativos. Já que não existe um estado "final" em uma computação infinita. O exemplo a seguir ilustra esse problema.

$$\begin{aligned} D[\![\mathit{while\ true}\{\mathit{complex\ procedure}\}]\!] &= D[\![\mathit{while\ true}\{\mathit{skip};\}]\!] \\ &= \lambda x : \mathit{Store}.\bot_{\mathit{Store}} \\ &= \bot_{\mathit{Store} \circ\!\!\rightarrow \mathit{Store}} \end{aligned}$$

Os problemas descritos acima podem ser resolvidos se utilizarmos funções de *Store* em $Store^{\infty}$ [4] para representar os programas escritos em uma linguagem imperativa. Neste caso estaríamos "armazenando" informações sobre os passos (*steps*) computacionais. Bem, na verdade, estamos resolvendo alguns problemas e criando outros, temos agora:

$$D'[\![x := 1; y := y + 1;]\!] \neq D'[\![x := 1; y := 2;]\!]$$

e mais ainda:

$$D'[\![\mathit{skip}; \mathit{skip};]\!] \neq D'[\![\mathit{skip};]\!]$$

Esses "problemas" relacionados com equivalência podem ser resolvidos utilizando uma noção de equivalência observacional [64] semelhante a usada em semântica operacional. Em resumo, temos que definir uma relação de equivalência para os elementos de $Store^{\infty}$. De qualquer forma, estaríamos perdendo a grande "vantagem"da semântica denotacional.

Problemas semelhantes aparecem quando modelamos linguagens concorrentes utilizando semântica denotacional. Neste caso, é necessário utilizar *Resumptions* [85] para modelar o paralelismo, e infelizmente, mais uma vez teremos que:

$$D'[\![\mathit{skip}; \mathit{skip};]\!] \neq D'[\![\mathit{skip};]\!]$$

Conseqüentemente, a noção de extensional de equivalência de semântica denotacional dificilmente poderá ser utilizada em linguagens "reais". Atualmente a maior parte das linguagens possui algum suporte a concorrência e paralelismo, e um grande número de aplicações descrevem processos que representam computações potencialmente infinitas, tais como sistemas reativos.

[4] $Store^{\infty}$ representa o conjunto de seqüências finitas e infinitas de *Store*.

Esta noção de equivalência não pode ser utilizada para justificar nenhuma otimização relevante (vide último exemplo).

No nosso ponto de vista, também não faz muito sentido basearmos procedimentos de análise inerentemente intencionais (envolvendo noções como passo computacional, transição, ...) como Interpretação Abstrata e Verificação de Modelos numa teoria extensional como semântica denotacional. Porém, existem diversos analisadores baseados em semântica denotacional [98, 34]. Uma crítica a esses trabalhos é descrita no capítulo sobre trabalhos correlatos.

4.9 *True Concurrency*

Nesta tese, todos os modelos de execução paralela são baseados em *interleaving*. Uma computação concorrente é representada por todas as possíveis combinações de "ações atômicas". Esta interpretação assume que existe a noção de "menor" passo computacional, que nada mais é do que a transição de *SOS* (i.e. *small steps*).

Mas pode-se argumentar que algumas linguagens não possuem a noção de "ação atômica" ou menor passo computacional. O programa $(x = 0 \parallel x = 1)$ modifica o valor da variável x em paralelo. Um processo modifica o valor da variável x para 0 e o outro processo para 1. A partir deste modelo de execução baseado em *interleaving*, concluiremos que o valor da variável x é igual a 0 ou 1 após a execução do programa. Contudo, isso não é verdade em linguagens de programação que não garantem que a atribuição é uma operação atômica. Nestas linguagens não é especificado o que acontece quando uma atribuição a uma mesma variável é realizada por processos paralelos [5]. Geralmente isto acontece porque a descrição da linguagem está em um nível de abstração elevado e independente de detalhes de *hardware*.

Por estes motivos, atualmente existe uma proliferação de modelos denominados de *true concurrency* ou *noninterleaving*. Nestes modelos, a concorrência é modelada sem impor uma ordem temporal ou causal. Por exemplo, na execução de $(x = 0 \parallel x = 1)$ nenhuma suposição é feita sobre a ordem em que $x = 0$ ou $x = 1$ serão executados. Estes modelos são baseados em *pomsets* [80] ou *event structures* [72]. Infelizmente, somente linguagens extremamente simples possuem modelos *true concurrency*.

De qualquer forma, para os nossos objetivos o modelo baseado em *interleaving* é mais do que suficiente. O problema relativo a atribuição concorrente pode ser resolvido através da adição de uma transição extra. Em resumo, ao executarmos $(x = 0 \parallel x = 1)$ teremos agora três *traces* possíveis:

1. $(x = 0 \parallel x = 1) \xrightarrow{\alpha_1} (\mathit{completed} \parallel x = 1) \xrightarrow{\alpha_2} (\mathit{completed} \parallel \mathit{completed}) \xrightarrow{\alpha_3} \mathit{completed}$
2. $(x = 0 \parallel x = 1) \xrightarrow{\alpha_1} (x = 0 \parallel \mathit{completed}) \xrightarrow{\alpha_2} (\mathit{completed} \parallel \mathit{completed}) \xrightarrow{\alpha_3} \mathit{completed}$
3. $(x = 0 \parallel x = 1) \xrightarrow{\alpha_1} (\mathit{completed} \parallel \mathit{completed}) \xrightarrow{\alpha_2} \mathit{completed}$

Os primeiros dois *traces* são os convencionais. No terceiro *trace* aparece o efeito de nossa modificação, que faz com que a operação de atribuição funcione de forma semelhante a um operador de sincronismo. Neste caso, estamos assumindo que as duas atribuições tenham acontecido "ao mesmo tempo" e portanto o valor de x é indefinido [6].

A partir da semântica descrita na Figura 4.10, precisamos apenas adicionar um novo "campo" no *label* que indica se alguma variável foi modificada. O predicado *var-updated* atualiza este

[5] Obviamente, processos paralelos podem modificar uma mesma variável sem maiores problemas, se for utilizado alguma forma de sincronismo (ex. semáforos).

[6] Representaremos o valor indefinido como $\top$.

$$\frac{\textit{set-store}(\alpha, Id, Val),\ \ \textit{var-updated}(\alpha, Id)}{Id := Val \xrightarrow{\alpha} \textit{completed}}$$

$$\frac{S_1 \xrightarrow{\alpha_1} S_1',\ \ S_2 \xrightarrow{\alpha_2} S_2',\ \ \textit{glue}(\alpha_1, \alpha_2, \alpha_3)}{S_1 \parallel S_2 \xrightarrow{\alpha_3} S_1' \parallel S_2'}$$

Figura 4.16: Exemplo de *SOS* modular

"campo" e pode ser definido como:

$$\textit{var-updated}(\alpha, var) \stackrel{\text{def}}{=} \alpha = \langle store_1, store_1', var \rangle$$

Além disso, temos um predicado denominado *glue* que pode ser definido como:

$$\textit{glue}(\alpha_1, \alpha_2, \alpha_3) \stackrel{\text{def}}{=} \begin{cases} \alpha_1 = \langle store_1, store_1', var \rangle \wedge \\ \alpha_2 = \langle store_1, store_2', var \rangle \wedge \\ \alpha_3 = \langle store_1, store_3', var \rangle \wedge \\ store_3' = store_1'[var \leftarrow \top] \wedge \\ store_3' = store_2'[var \leftarrow \top] \end{cases}$$

Este predicado "cola" dois *labels* (α_1 e α_2), se estes possuem o mesmo *store* inicial e a única diferença do *store* final é o valor da variável *var*. O novo *label* (α_3) é o resultado da "colagem". Com a utilização destes predicados podemos especificar a nossa modificação. Em resumo, precisamos modificar a regra de atribuição e adicionar uma nova regra para o operador $\parallel$. A Figura 4.16 contém estas modificações.

É importante salientar que a atualização concorrente [7] de uma variável é, em geral, um erro de programação. Logo, a nossa técnica pode ser utilizada para detectar este tipo de situação.

4.10 Conclusão

SOS é uma forma extremamente simples de dar a semântica de linguagens de programação. Ao utilizarmos técnicas de encapsulamento de dados, conseguimos definir descrições modulares que podem ser reutilizadas. *SOS* também se encaixa perfeitamente com as teorias de Interpretação Abstrata e Verificação de Modelos, já que no coração de *SOS* está a noção de transição, que é fundamental para a construção dos mapas de estados (sistemas de transição). A modularidade obtida também será de grande utilidade em nosso trabalho, permitindo o desenvolvimento modular de analisadores e verificadores de código.

4.10.1 Contribuições

- Visão pragmática de *Modular SOS*, i.e. *Modular SOS* = *SOS* + *ADTs* [8];
- Relacionamento entre *labels* e *monads*;
- Como implementar *Modular SOS*;

[7] Atualização de uma variável por dois processos diferentes sem a utilização de uma forma de sincronismo (ex.: semáforos).

[8] *ADT* é uma sigla para designar tipos abstratos de dados.

- Justificativa do porquê é desinteressante se utilizar semântica denotacional para construir analisadores de código
- Técnica de como lidar com linguagens onde não existam a noção de "passo atômico". A partir dessa técnica foi mostrado como construir um analisador que detecta atualizações concorrentes.

Capítulo 5

O "Framework" de Análise e Verificação

5.1 Introdução

O nosso "framework" para a construção de analisadores de programas é baseado nas teorias de *Interpretação Abstrata* [27, 25], *Verificação de Modelos* [19, 20] e *Structural Operational Semantics* [1]. Para construirmos um novo analisador, devemos especificar um interpretador (abstrato ou não) para os programas a serem analisados. A especificação desse interpretador é feita através de uma linguagem lógica, que fornece um nível de abstração semelhante ao de *SOS*. Apesar de realizarmos análises utilizando especificações *SOS* (interpretadores *SOS-like*), isto não é uma regra em nosso "framework". A nossa única exigência é a existência de uma noção de transição na especificação do interpretador. Estas transições são utilizadas para navegar pelo mapa de estados.

O nosso "framework" fornece suporte a construção de analisadores de fluxo de dados [2], verificadores de código (*Verificadores de Modelo*) e depuradores estáticos. O nosso "framework" suporta diferentes tipos de técnica de análise. Neste capítulo, mostraremos como estas técnicas são *implementadas*.

Os passos (Figura 5.1) para a definição de um novo analisador são os seguintes:

1. Criar um analisador sintático para a linguagem cujos programas serão analisados. O analisador sintático deve produzir uma árvore de sintaxe abstrata.

2. *[Opcional]* Transformar a árvore de sintaxe abstrata em uma representação que seja satisfatória para realizar a(s) análise(s). Denominamos esta representação de meta-linguagem ou representação intermediária.

3. *[Opcional]* Especificar a semântica operacional da meta-linguagem ou da própria linguagem, se o passo anterior foi ignorado. Isto é, especifique um interpretador em nossa linguagem lógica.

4. Especificar o interpretador, abstrato ou não, que será utilizado para realizar a análise. A correção do analisador é justificada, quando necessário, em relação a semântica operacional da linguagem, mostrando que o interpretador utilizado pela análise é uma aproximação *segura* do interpretador *concreto* da linguagem (Capítulo 2).

[1] Um módulo de *Constraint Solving* também está disponível, mas este não é parte significativa de nosso "framework".

[2] Analisadores existentes em compiladores.

[3] AST da represetação intermediária associada ao programa.

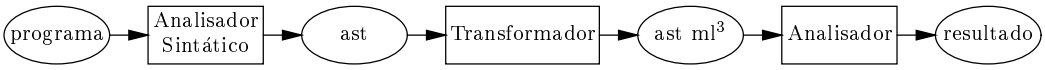

Figura 5.1: O processo de análise

5.2 Analisador Sintático

O nosso "framework" não provê nenhum suporte a geração de *analisadores sintáticos* para as linguagens a serem analisadas. Assumimos que a árvore de sintaxe abstrata de um programa é gerada por um módulo a parte. Este módulo pode ser construído usando ferramentas como *YACC* [59], *Antlr* [76], *TXL* [24] ou *Draco* [69, 58]. Nos exemplos apresentados nessa tese, a ferramenta *YACC* foi utilizada para gerar os analisadores sintáticos.

5.3 Meta-Linguagens

Em nosso "framework" para análise de programas, estimulamos o uso de meta-linguagens. Através do uso de meta-linguagens podemos simplificar em muito a definição de analisadores. Por exemplo, durante a conversão para meta-linguagem podemos remover o *açúcar sintático* contido na linguagem e transformar comandos ou conceitos complexos em um seqüência ou conjunto destes mais simples. Além disso, se a meta-linguagem for definida de forma modular, podemos reutilizar/estender meta-linguagens ou fragmentos destas na definição de novos analisadores. Por exemplo, uma meta-linguagem utilizada para analisar programas *C* pode ser adaptada, ou até mesmo utilizada diretamente, para analisar programas *Pascal*. Obviamente, também é possível realizar diferentes análises sobre a mesma meta-linguagem. Portanto, em nosso "framework" o principal elemento de reutilização são as meta-linguagens. Ao reutilizarmos uma meta-linguagem, estamos reutilizando os analisadores definidos para a mesma. Outra forma de conseguirmos reutilizar analisadores é através da transformação de uma meta-linguagem em outra. Considere o caso de termos uma meta-linguagem $\mathcal{L}_1$ e um analisador $\mathcal{A}_1$ para a mesma, e uma meta-linguagem $\mathcal{L}_2$ com um analisador $\mathcal{A}_2$. Neste caso é comum ser mais simples transformar $\mathcal{L}_2$ em $\mathcal{L}_1$, do que implementar o analisador $\mathcal{A}_1$ para $\mathcal{L}_2$. Contudo, a implementação desta transformação nem sempre é possível, ou mais simples do que a reconstruição do analisador. Para exemplificar esta situação, basta considerarmos o caso de $\mathcal{L}_1$ ser uma linguagem imperativa e $\mathcal{L}_2$ uma linguagem lógica.

Esta visão de meta-linguagens como elementos de reutilização coincide com a proposta dos ambientes *Draco* [69, 70, 58] e *DMS* [9], onde nossas meta-linguagens seriam denominadas de domínios semânticos.

Do ponto de vista pragmático, a utilização de meta-linguagens coincide com a prática de construção de compiladores, onde todas as análises não realizadas sobre representações intermediárias. Na construção de compiladores, a utilização de representações intermediárias não é só uma decisão de projeto, mas também uma necessidade, porque algumas análises só fazem sentido quando considerando a representação intermediária [68].

Para especificarmos a transformação de uma linguagem para uma meta-linguagem, ou de uma meta-linguagem para outra meta-linguagem, utilizamos a mesma linguagem lógica utilizada para especificar os analisadores. Esta decisão é justificada na literatura, onde *Prolog* já foi utilizado com sucesso na definição de transformações [94], implementação de compiladores [102] e gramáticas de atributos [90]. Pode-se argumentar que os transformadores não terão uma eficiência satisfatória devido ao uso de uma linguagem lógica, entretanto, este não é o caso, conforme será justificado nos próximos capítulos.

5.4 Especificando a Semântica

A especificação da semântica operacional é feita através da construção de um interpretador em nossa linguagem lógica. Este interpretador deve possuir uma noção de transição, e uma computação deve ser formada por uma seqüência finita ou (potencialmente) infinita de transições. A função principal do interpretador é ser um "modelo" para a definição dos analisadores e verificadores. Acreditamos que é muito mais simples definir um analisador/verificador depois de termos implementado um interpretador (concreto) para a linguagem. Além de ser possível justificar que as aproximações utilizadas no analisador são seguras em relação a semântica da linguagem (i.e. interpretador concreto).

Um interpretador concreto também serve para observarmos situações de erro detectadas por um verificador. Por exemplo, suponha que o verificador tenha detectado uma situação de erro e tenha produzido um *trace* que mostra como situação acontece, logo podemos utilizar o interpretador concreto para executarmos o *trace* passo a passo.

No caso de programas que possuam um mapa de estados finito, podemos utilizar, também, o interpretador concreto para realizarmos as análises, sendo necessário apenas utilizarmos a técnica simples descrita no Capítulo 3. Após a geração do mapa de estados, utilizamos os algoritmos de verificação de propriedade (Seção 3.4.2). No caso de programas que possuam mapa de estados infinito, o interpretador concreto também pode ser utilizado em conjunto com a técnica de execução livre, ou até mesmo com a técnica de *supertrace*, descrita no Capítulo 3.

A especificação do interpretador é bastante semelhante ao programa *Prolog* mostrado no capítulo anterior. A diferença é que no lugar de *Prolog* utilizamos a nossa própria linguagem lógica, que denominamos de *PAN*. Também não é necessário seguir exatamente a estrutura de *SOS*. Em *SOS*, por exemplo, os termos da linguagem funcionam como uma espécie de *instruction pointer*, apesar de ser uma forma elegante de implementar um interpretador, esta pode gerar problemas de eficiência, por isso é muito comum substituir o uso dos termos por um *instruction pointer* ou algo parecido. Independentemente de estarmos utilizando a estrutura de *SOS* ou não, é interessante termos descrições modulares, logo devemos sempre utilizar tipos abstratos de dados para encapsular o estado da computação.

5.5 Especificando Analisadores

Em geral, para definirmos um analisador, devemos aproximar a semântica operacional da linguagem, i.e. o seu interpretador concreto. O objetivo dessas aproximações é garantir a terminação da análise, ou melhorar a sua eficiência, ou abstrair comportamentos irrelevantes, possibilitando inclusive a análise e verificação de programas e/ou especificações com mapas de estados infinitos, fato esse que quase sempre acontece na prática.

Este processo de aproximação consiste em substituir regras da semântica concreta por versões aproximadas. Em alguns casos também são adicionadas novas regras. Suponha que estejamos implementando um analisador de sinais para a linguagem imperativa descrita no capítulo anterior. Neste caso, uma variável booleana pode assumir os valores *true*, *false* e $\top$ (representado o valor "não sei"). A presença do valor $\top$ é necessária, já que apesar de sabermos com certeza o valor que algumas expressões booleanas (ex.: $neq < pos$) irão assumir, *não* sabemos qual o valor de outras (ex.: $pos < pos$). Podemos adicionar, assim, duas novas regras que representam o comportamento do comando *if* na presença do valor $\top$.

$$if\ \top\ then\ S_1\ else\ S_2 \xrightarrow{i} S_1$$

$$if\ \top\ then\ S_1\ else\ S_2 \xrightarrow{i} S_2$$

5.5.1 *SOS* e a terminação do analisador

Em *SOS* há um problema adicional. As regras *SOS* podem gerar sintaxe nova [4] em tempo de execução. Ou seja, as derivações *SOS* podem não ser semi-composicionais [5] e por este motivo, elas podem "atrapalhar" a terminação do analisador. Considere a descrição *SOS* da Figura 5.2. Esta é a descrição da semântica de uma linguagem simples que possui os comandos *unfolding*, *unfold*, *skip*, "+" e ";". Os dois primeiros servem para criar recursões, o terceiro é o tradicional comando "nulo", e os dois últimos representam a escolha não determinística e o comando de composição seqüencial, respectivamente. Ao considerarmos esta semântica e o termo $unfolding(unfold; skip)$, teremos a seguinte sequência infinita de transições:

$$
\begin{aligned}
&unfolding(unfold; skip) \xrightarrow{i} \\
&(unfold; skip) \ @ \ (unfold; skip) \xrightarrow{i} \\
&(unfold; skip; skip) \ @ \ (unfold; skip) \xrightarrow{i} \\
&(unfold; skip; skip; skip) \ @ \ (unfold; skip) \xrightarrow{i} \\
&(unfold; skip; skip; skip; skip) \ @ \ (unfold; skip) \xrightarrow{i} \\
&\ldots
\end{aligned}
$$

O analisador nunca irá terminar, porque este nunca encontrará um estado igual a um estado já visitado. Por outro lado, o analisador termina ao analisar o termo $unfolding(skip; unfold)$, gerando a seguinte sequência de transições.

$$
\begin{aligned}
&unfolding(skip; unfold) \xrightarrow{i} \\
&(skip; unfold) \ @ \ (skip; unfold) \xrightarrow{i} \\
&(completed; unfold) \ @ \ (skip; unfold) \xrightarrow{i} \\
&(unfold) \ @ \ (skip; unfold) \xrightarrow{i} \\
&(skip; unfold) \ @ \ (skip; unfold) \quad \textit{Loop detectado!!!}
\end{aligned}
$$

Apesar deste exemplo de descrição *SOS* parecer "exótico", este pode ser considerado como um fragmento da semântica operacional da *Action Notation* [66].

Para resolvermos este problema, temos que abstrair a sintaxe abstrata. Ou seja, temos que definir uma forma de "compactar" a árvore de sintaxe abstrata. Esta compactação é melhor descrita graficamente, onde representamos os termos como árvores. A partir dessa representação em árvore, compactamos árvores que estejam possivelmente "divergindo". Podemos considerar uma árvore "divergente", aquela cuja altura esta aumentando após um número x de comandos *unfold*. A "compactação" gera um elemento que representa uma ou mais árvores. Este elemento é composto por termos da linguagem, e *nodos* (*or-nodes*) que representam alternativas. Considere a Figura 5.3, o elemento descrito representa duas árvores: a árvore composta apenas pelo comando *skip* e a árvore composta pelo valor *completed*. O elemento descrito na Figura 5.4 representa a sequência infinita de termos:

$$
\begin{aligned}
&unfold; skip \\
&unfold; skip; skip \\
&unfold; skip; skip; skip \\
&\ldots
\end{aligned}
$$

Nesta técnica a terminação da análise é garantida. A principal vantagem desta, é poder ser utilizada por qualquer analisador, cuja a descrição *SOS* não é semi-composicional. Mas fica

[4] Termos novos que não são subtermos do programa.

[5] Uma derivação é semi-composicional se todos os termos que aparecem em uma derivação são subtermos do programa original.

$$unfolding\ A \xrightarrow{i} A \ @\ A$$

$$completed\ @\ A \xrightarrow{i} completed$$

$$\frac{set\text{-}unfolding(\alpha, A_0, \alpha'),\ A \xrightarrow{\alpha'} A'}{A\ @\ A_0 \xrightarrow{\alpha} A'\ @\ A_0}$$

$$\frac{get\text{-}unfolding(\alpha, A_0)}{unfold \xrightarrow{\alpha} A_0}$$

$$skip \xrightarrow{i} completed$$

$$A_1 + A_2 \xrightarrow{\alpha} A_1$$

$$A_1 + A_2 \xrightarrow{\alpha} A_2$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1; S_2 \xrightarrow{\alpha} S_1'; S_2} \qquad completed\ ; S_2 \xrightarrow{i} S_2$$

Figura 5.2: Exemplo de descrição não semi-composicional

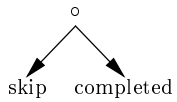

Figura 5.3: Representação de ASTs - Exemplo 1

claro que esta técnica compromete a eficiência do analisador, porque a cada comando *unfold* é realizada uma detecção de divergência, e em caso afirmativo é realizada a "compactação" do termo. Uma técnica simples para detectarmos divergência é verificarmos se o tamanho do termo continua crescendo após um número x de *unfold's*.

É importante salientar que ao utilizarmos esta técnica adicionamos uma nova fonte de não determinismo, devido aos *or-nodes* (representados como ∘ nas Figuras 5.3 e 5.4).

5.6 Tipos de análise suportados

O nosso "framework" provê suporte para a construção de diferentes tipos de analisador. Cada tipo de analisador possui certas peculiaridades, que serão expostas a seguir. Enfim, estamos sugerindo uma abordagem para a construção de cada tipo de analisador.

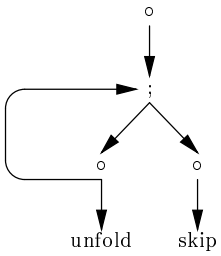

Figura 5.4: Representação de ASTs - Exemplo 2

5.6.1 Verificação de Programas Simples

No caso de estarmos interessados em construir um verificador para programas simples, onde o termo "programa simples" significa que o mapa de estados do programa [6] é finito. Neste caso, podemos utilizar a abordagem ingênua, onde construímos o mapa de estados explicitamente. A geração do mapa de estados pode ser feita utilizando a semântica operacional da linguagem (i.e. do interpretador concreto). Após termos gerado o mapa de estados, podemos utilizar os algoritmos de verificação de propriedades apresentados no Capítulo 3.

A implementação deste tipo de analisador é simples, consistindo apenas em modificar um interpretador, para este armazenar os estados da computação.

No caso de programas com mapa de estados finito muito grande, as técnicas de verificação de mapa de estados infinito definidas a seguir podem ser utilizadas.

5.6.2 Verificação de Programas Complexos

Neste caso, estamos interessados em construir um analisador para programas com mapas de estado potencialmente infinitos ou muito grandes. Existem diversas abordagens para este tipo de situação.

A mais simples de todas consiste em utilizarmos a abordagem de execução livre descrita no Capítulo 3. Neste caso, precisamos fazer apenas pequenas modificações em nosso interpretador concreto, para que este cheque as propriedades desejadas enquanto realiza a "interpretação". Como já mencionado, nem todas as propriedades especificadas em uma lógica modal podem ser verificadas. Desta forma, utilizamos uma abordagem pragmática, onde permitimos a adição de marcações no código do programa que correspondem as propriedades a serem checadas. Permitimos o seguinte conjunto de propriedades:

assert(condition) Quando o analisador encontra esta "marcação", este verifica se o estado corrente satisfaz a condição especificada, caso contrário uma mensagem de erro é gerada.

Assert(condition) Verifica se o estado atual e todos os seus sucessores satisfazem a condição especificada. Esta marcação pode ser utilizada no início do programa para verificar se todos os estados satisfazem a uma determinada condição.

markProgress Esta marcação sinaliza ao interpretador que uma situação de progresso foi encontrada. A cada transição, em um determinado caminho, o interpretador incrementa a variável *numSteps*. Se o valor for maior que um valor x especificado pelo o usuário, o interpretador gera um erro indicando que o código possivelmente está em um *dead-lock* ou *live-lock*. Ao encontrar a marcação *markProgress* o interpretador modifica o valor da variável *numSteps* para zero. Esta marcação tem a função de indicar "progressos" na execução do programa. É óbvio que as mensagens de erro geradas podem não ser válidas, visto que o valor x pode ser muito pequeno.

As variações da técnica de obtenção do estado sucessor descritas na Seção 3.5.3 também podem ser utilizadas.

As técnicas apresentadas, até agora, podem ser implementadas com facilidade utilizando-se diretamente a semântica operacional da linguagem. Se esta semântica foi definida de forma modular, a estrutura de ambos analisadores é, então, modular e *reutilizável*.

[6] O mapa de estados de programas é em geral (potencialmente) infinito, por esta razão esta abordagem se aplica principalmente a especificações. Também podemos entender o termo "finito" como significando "uma quantidade tratável".

5.6.2.1 Meta-linguagens genéricas

A abordagem baseada em execução livre permite que especifiquemos um verificador para uma meta-linguagem genérica, que pode ser reutilizada para descrevermos um grande número de linguagens. Por exemplo, o Apêndice E contém a semântica operacional de algumas facetas da *Action Notation* [66] codificada na linguagem *PAN*. Logo, para definirmos um verificador baseado em execução livre para uma nova linguagem de programação, precisamos apenas mapear a linguagem de programação para a *Action Notation*. Este mapeamento nada mais é do que a semântica de ações da linguagem de programação. Desta maneira, conseguimos gerar automaticamente um verificador a partir da semântica de ações de uma linguagem de programação. É lógico que podemos obter um verificador mais eficiente se utilizarmos uma meta-linguagem especialmente desenvolvida para a linguagem que estamos interessados em analisar. Por outro lado, obter um verificador automaticamente a partir da semântica de ações pode ser extremamente útil no caso de *DSLs* [7].

5.6.2.2 Execução Livre + Aproximações

O método de execução livre pode, também, ser utilizado em conjunto com aproximações, principalmente no caso em que as aproximações não garantem a terminação do analisador. Estas aproximações funcionam apenas como um mecanismo para diminuir a quantidade de estados. Neste caso, precisamos apenas justificar que as regras modificadas ou adicionadas são aproximações seguras. Esta combinação do método de execução livre e aproximações é útil principalmente no caso em que descrever aproximações que garantam a terminação é muito complicado, ou quando as aproximações que garantem a terminação são muito imprecisas. É extremamente complexo, por exemplo, definir aproximações que garantam a terminação de verificadores que trabalham sobre a *Action Notation*.

5.6.2.3 Forçando a Terminação

A técnica de *supertrace* também pode ser vista como um método para forçar a terminação do método de execução livre. A única modificação que precisamos fazer no verificador é adicionar uma função que calcula o valor de hash de um estado e um *array de bits* que indica se um estado já foi visitado ou não.

Existem outras formas de garantir a terminação. Uma delas consiste em armazenar explicitamente os estados visitados e utilizar aproximações para garantir a terminação. Infelizmente, essa técnica só pode ser aplicada em programas pequenos ou especificações, porque esta consome uma quantidade muito grande de memória. Esta técnica pode até ser adaptada para programas maiores, se utilizarmos mais aproximações. Entretanto, quanto mais aproximações utilizarmos, menos "erros verdadeiros" conseguiremos identificar. Ao adicionarmos aproximações seguras, em geral, estamos adicionando comportamentos que não existiam no programa original. Uma aproximação segura garante apenas a reprodução de todos os comportamentos do programa original, não restringindo, portanto, a existência de comportamentos adicionais. Desta maneira, se um erro não é identificado, temos "certeza" que este não ocorre no programa original. Por outro lado, se detectamos um erro, existe a possibilidade deste não ser realmente um "erro verdadeiro". Este pode ter aparecido pelo fato das aproximações estarem adicionando comportamentos espúrios. Por isso, todo o erro detectado por um verificador que usa aproximações deve ser verificado pelo usuário. O uso de aproximações também mascara alguns erros, por exemplo se quisermos verificar a propriedade "*existe* um *trace* t para o qual o programa $\mathcal{X}$ termina". Todavia, não podemos garantir que o verificador irá responder corretamente esta pergunta, já

[7] Domain Specific Languages.

que as aproximações podem ter adicionado um *trace* que termina. O importante é termos em mente que não estamos tentando provar correção e sim detectar erros. Seguindo esta linha de pensamento, podemos fazer uma analogia entre o processo de abstração (uso de aproximações) e o desenvolvimento de modelos (protótipos) em outras disciplinas de engenharia. Por exemplo, quando produz-se um protótipo de um carro, avião ou represa, a função é identificar falhas de projeto e validar idéias. De forma alguma, o correto "funcionamento" do modelo garante o correto "funcionamento" do "produto final". Por este motivo, insistimos que o propósito de métodos formais é "detectar erros de projeto" e não provar correção, que dentro de nosso ponto de vista é completamente "utópico".

Além das técnicas de *supertrace* e aproximações, podemos garantir a terminação realizando uma análise *depth-first*, em que armazenamos apenas o caminho (*trace*) corrente. Neste caso, podemos utilizar uma função que identifica se um *trace* está divergindo ou não. Obviamente, esta resposta pode ser apenas aproximada, visto que verificar divergência é indecidível. Podemos citar os seguintes exemplos de função de detecção de divergência:

Número de Passos A função mais simples de detecção de divergência apenas verifica se o número de passos em *trace* é maior que um valor pré determinado.

$$\mathcal{F}(trace) \stackrel{\text{def}}{=} (length(trace) > numMaxSteps)$$

Hashing O método de *supertrace* pode ser adaptado nesta técnica.

$$\mathcal{F}(trace) \stackrel{\text{def}}{=} \exists s_1, s_2 \in trace \cdot hash(s_1) = hash(s_2)$$

Este método é mais preciso que o *supertrace* tradicional, porque este somente não visita um estado, se já existir no mesmo *trace* (caminho) um estado com mesmo valor de *hash*. Enquanto que o *supertrace* não visita um estado se já existir no *grafo* um estado como mesmo valor de *hash*. Podemos melhorar ainda mais a precisão desta função de detecção, se obrigarmos a existir pelo menos um número n de estados cujo valor de *hash* seja igual. Temos, então, algo como:

$$\mathcal{F}(trace) \stackrel{\text{def}}{=} \exists s_1, \ldots, s_n \in trace \cdot hash(s_1) = \ldots = hash(s_n)$$

Peso Podemos definir uma função de peso para o estado que indica o "potencial" de divergência de um estado. A partir desta função podemos definir a seguinte função de detecção:

$$\mathcal{F}(trace) \stackrel{\text{def}}{=} \exists s_1, \ldots, s_n \in trace \cdot weight(s_1) < \ldots < weight(s_n)$$

Ou seja, $\mathcal{F}$ detecta divergência se existe uma sequência de estados de tamanho n cujo o "potencial" de divergência está aumentando. Podemos definir, por exemplo, uma função de peso que retorna o tamanho do termo associado ao estado. Esta função pode ser utilizada para garantir a terminação do exemplo envolvendo os comandos *unfolding* e *unfold*.

Conjunção Se $\mathcal{F}_1$ e $\mathcal{F}_2$ são funções de detecção de divergência, então

$$\mathcal{F}(trace) \stackrel{\text{def}}{=} \mathcal{F}_1(trace) \wedge \mathcal{F}_2(trace)$$

também é uma função de detecção de divergência mais precisa [8] que $\mathcal{F}_1$ e $\mathcal{F}_2$.

[8] $\mathcal{F}$ é mais precisa que $\mathcal{F}'$, se e somente se, $\forall trace \cdot \mathcal{F}(trace) \Rightarrow \mathcal{F}'(trace)$. Se um *trace* é erroneamente detectado como divergente por $\mathcal{F}$, então este também será detectado erroneamente por $\mathcal{F}'$.

O método de detecção de divergência pode ser aprimorado mudando o comportamento do verificador no momento de detecção de uma divergência. Ao invés de parar de verificar o *trace* corrente, podemos fazer o verificador começar a utilizar aproximações menos precisas para garantir a terminação da análise. Ou melhor, podemos utilizar uma sequência de pares $\langle \mathcal{F}_i, \mathcal{A}_i \rangle$, onde os $\mathcal{F}_i$ são detectores de divergência e os $\mathcal{A}_i$ são "aproximadores" semânticos, e se $i < j$, então $\mathcal{F}_j$ é mais preciso que $\mathcal{F}_i$ e $\mathcal{A}_i$ é mais preciso que $\mathcal{A}_j$. O verificador começa utilizando o par $\langle \mathcal{F}_i, \mathcal{A}_i \rangle$ onde $i = 0$, e sempre que $\mathcal{F}_i$ detectar uma divergência, então o verificador passa a usar o par $\langle \mathcal{F}_{i+1}, \mathcal{A}_{i+1} \rangle$, onde $\mathcal{A}_{i+1}$ força a terminação de forma mais "forte" que $\mathcal{A}_i$, pois é menos preciso. O último elemento desta sequência de pares pode ser $\langle \mathcal{F}_n, \mathcal{A}_n \rangle$, onde $\mathcal{F}_n(trace) \stackrel{\text{def}}{=} false$ nunca detecta divergência e $\mathcal{A}_n$ é uma aproximação bastante imprecisa que garante a terminação. Por exemplo, suponha que estejamos desenvolvendo um verificador para a linguagem imperativa simples descrita no Capítulo 4, usando esta técnica para forçar a terminação podemos definir a seguinte seqüência de pares:

- $\mathcal{F}_0$ verifica se o número de passos é maior que um valor n, e $\mathcal{A}_0$ não realiza nenhuma aproximação
- $\mathcal{F}_1$ utiliza *hashing* e verifica se o número de passos é maior que um valor m, tal que $m > n$. $\mathcal{A}_1$ utiliza aproximação de intervalos e o operador *join*.
- $\mathcal{F}_2$ "retorna" sempre *false*, i.e. nunca detecta divergência, e $\mathcal{A}_2$ pode utilizar aproximação de sinais, ou intervalos mais o operador de *widening*, garantindo, desta maneira, a terminação da análise.

É importante salientar que o uso dos operadores *join* e *widening* é compatível com o método de análise *depth-first*. Não é possível utilizar estes operadores com o método de execução livre, se este não armazenar explicitamente os estados contidos no caminho corrente. A informação contida no caminho corrente [9] é suficiente para aplicarmos os operadores *join* e *widening* (vide os algoritmos das Figura 2.17 e 2.19).

5.6.2.4 Propriedades Temporais

Propriedades temporais podem ser utilizadas para verificar condições que envolvem *traces*, e não somente estados [10]. Por exemplo, propriedades do tipo "se um estado s_1 satisfaz a condição P então eventualmente um estado s_2 sucessor de s_1 irá satisfazer a condição Q" (esta propriedade pode ser representada, em lógica temporal linear, como $\Box(P \rightarrow \Diamond Q)$). Verificar este tipo de propriedade utilizando a abordagem ingênua é elementar, mas esta abordagem não se aplica a programas com um número de estados infinito ou muito grande, nestes casos o mapa de estados não é gerado explicitamente. É interessante adaptarmos as técnicas descritas na seção anterior [11] para detectarmos este tipo de propriedade. O segredo é conseguir checar a propriedade durante a "interpretação". Primeiro, devemos notar que não estamos realmente interessados em verificar a propriedade, mas sim situações em que a mesma não seja satisfeita (situações de erro). A partir dessa observação, é possível verificar se um *trace* não satisfaz uma propriedade temporal, quando ele é homomórfico a um *trace* que descreve a situação de erro [46]. O sistema de transição da Figura 5.5, por exemplo, representa abstratamente todos os *traces* que não satisfazem a condição $\Box(P \rightarrow \Diamond Q)$. Este sistema de transição pode ser "entendido" da seguinte forma:

[9] Se um verificador não armazena o caminho corrente, então este terá que ser reinicializado sempre que for necessário analisar um caminho alternativo, vito que não será possível realizar um *backtracking*. Este é o caso da ferramenta *Verisoft* (Seção 10.5).

[10] Os marcadores *assert* e *Assert* podem ser vistos como descrevendo propriedades temporais simples. A marcação $assert(P)$ é equivalente a propriedade P, e a marcação $Assert(P)$ é equivalente a propriedade $\Box P$.

[11] Isto é, execução livre, *supertrace* e "*depth-first search* + funções de detecção de divergência".

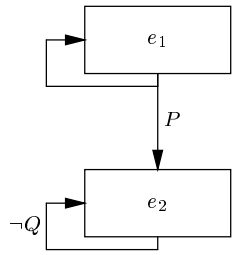

Figura 5.5: Abstração dos *traces* que não satisfazem $\Box(P \rightarrow \Diamond Q)$

- inicialmente a condição P não é satisfeita, então o processo permanece no estado e_1 até P ser satisfeita;
- ao encontrar um estado onde P é satisfeita, temos duas opções, continuar no estado e_1 ou ir para o estado e_2;
- no estado e_2 a única transição possível é realizada se Q não for satisfeita.

Caso nenhuma transição possa ser realizada a situação de erro não pode ocorrer.

Para verificarmos uma propriedade temporal, associamos um estado do programa a um estado do sistema de transição $\mathcal{ERROR}$ que representa a situação de erro. Prosseguimos a verificação enquanto é possível manter esta relação. Quando o programa realiza uma transição que não pode ser representada por $\mathcal{ERROR}$, o verificador interrompe a análise do *trace* corrente, já que a situação de erro não pode ocorrer, porque não existe um homomorfismo entre o mapa de estado do programa e $\mathcal{ERROR}$. Este método aumenta o espaço de busca, já que $\mathcal{ERROR}$ é não determinístico. Suponha que estejamos num estado s_n do programa, e este está associado ao estado e_1 do exemplo da Figura 5.5 (esta associação será representada como $\langle s_n, e_1 \rangle$). Suponha também que a condição P é satisfeita em s_n, então o verificador deve considerar duas novas alternativas $\langle s_{n+1}, e_1 \rangle$ e $\langle s_{n+1}, e_2 \rangle$. Por este motivo, o número de estados visitados pode, então, no pior caso ser multiplicado pelo número de estados de $\mathcal{ERROR}$.

5.6.2.5 Marcadores *Assure*

Como já foi dito anteriormente, o uso de aproximações pode gerar comportamentos não desejados. Para lidarmos com este tipo de situação, definimos o marcador *assure*. Este marcador funciona de forma semelhante ao *assert*. Quando o analisador encontra um *assert*, este verifica se o estado corrente satisfaz a condição especificada, caso contrário uma mensagem de erro é gerada. No caso do *assure*, o analisador assume que a condição é verdadeira. Se um estado não satisfaz a condição, o *trace* corrente deve ser descartado, o analisador assume que o *trace* é fruto das imprecisões geradas pelo uso de aproximações. O marcador *assure* permite que as imprecisões geradas pelas aproximações sejam minimizadas. É interessante, também, a utilização destes marcadores por analisadores contidos em compiladores. Estes analisadores podem utilizar este tipo de marcador para melhorar os resultados da análise e gerar código mais eficiente. Alguns compiladores possuem marcações que podem ser consideradas como "instâncias" do marcador *assure*. Por exemplo, o compilador da linguagem funcional *Clean* [77] possui marcações que indicam se um parâmetro é *strict* [12] ou não. Estas marcações auxiliam o *strict analyzer* [13].

[12] Um parâmetro de uma função f é chamado *strict* se $f(\perp) = \perp$, onde $\perp$ está representando as computações "não terminantes".

[13] *Clean* é uma linguagem funcional *lazy*. Mas a avaliação *lazy* é menos eficiente que a avaliação *eager*. Desta forma, substituir o mecanismo de avaliação (de *lazy* para *eager*) de algumas funções é uma otimização muito comum. Entretanto, esta otimização só é válida se o parâmetro for *strict*.

É claro que o uso do marcador *assure* deve ser cuidadoso, visto que se colocarmos uma condição "falsa", poderemos estar "mascarando" diversos erros. No caso de compiladores, o uso incorreto do marcador pode modificar a semântica do programa. Na linguagem *Clean*, por exemplo, se marcarmos um parâmetro que não é *strict* como *strict*, poderemos estar transformando uma computação que termina em uma que não termina.

Da mesma forma que o marcador *assert* possui uma versão temporal simples (*Assert*), podemos, também, definir o marcador *Assure*, que informa ao analisador uma condição que deve ser válida para o estado atual e todos os seus sucessores. Se o estado atual ou algum de seus sucessores não satisfizer a condição, o *trace* corrente é descartado pelo analisador.

A analogia entre o *assert* e o *assure* pode ser extendida através da utilização de propriedades temporais que temos certeza que são válidas. Neste caso, o verificador funciona de forma semelhante ao descrito na seção anterior. Porém, ao invés de descrevermos uma situação de erro, descrevemos uma situação que temos certeza que ocorre. Em suma, associamos um estado do programa a um estado do sistema de transição $\mathcal{ASSURE}$ que representa a situação que tem que ocorrer [14].

Os marcadores *assure* são utilizados para descartar *traces* provenientes de imprecisões. Podemos, também, utilizar estes marcadores para selecionar um subconjunto de *traces* "interessantes". Chamamos de *traces* "interessantes", aqueles que possuem um funcionamento obscuro. O marcador *assure* funciona como um *filtro*, que informa ao analisador quais os *traces* que estamos interessados em verificar.

5.6.3 Analisadores

Compiladores utilizam informações geradas por analisadores de código para aplicar otimizações em programas. Estes analisadores devem garantir a terminação da análise, porque não faz sentido, a princípio, um compilador que entra em *loop*. Por este motivo, os analisadores utilizados por compiladores utilizam aproximações para garantir a terminação da análise. Ao contrário dos verificadores de código, estes analisadores não podem simplesmente interromper o processamento, porque o compilador não pode utilizar informações de uma análise incompleta. Informações de uma análise incompleta não são seguras, já que não contêm informações sobre todos os possíveis comportamentos do programa. A Figura 5.6 contém as principais diferenças entre verificadores e analisadores de código.

Os algoritmos utilizados na construção de *analisadores de fluxo de dados* são semelhantes aos apresentados na Seção 2.6.

Outro requisito é a eficiência. As análises em geral são "grosseiras", pelo fato de existir a "cultura" de que um compilador deva ser eficiente. Dentro do nosso ponto de vista tal argumento não procede, porque podemos gerar código não otimizado durante a fase de desenvolvimento do programa, e gerar uma cópia otimizada para a versão de produção. Logo, a princípio, faz sentido termos tempos de compilação altíssimos, se o código gerado for realmente mais otimizado.

Se o objetivo for extrair informações de *DSLs*, a análise pode até ser baseada na semântica da linguagem, entretanto, se o objetivo for analisar linguagens de programação como *C* e *Pascal*, dificilmente conseguiremos ter uma eficiência razoável para o analisador. Isto acontecerá principalmente se estivermos interessados em análises interprocedurais [15]. Nestes casos, consideramos que a análise deve ser baseada numa descrição de baixo nível (meta-linguagem), utilizando por exemplo *instruction pointer*, ao invés de termos.

Neste tipo de análise é muito comum a utilização do operador *join* para "juntar" as informações associadas a um determinado ponto de controle. Diferentemente do verificador que pode

[14] O sistema de transição $\mathcal{ASSURE}$ é o análogo ao sistema de transição $\mathcal{ERROR}$.

[15] A maior parte dos compiladores não realiza análise interprocedural devido a problemas de *performace* e complexidade da análise.

- Verificadores

    1. O objetivo é detectar erros.
    2. Abandona um *trace* ao detectar divergência. Salientamos que este procedimento só faz sentido em programas/especificações contendo um número de estados potencialmente infinito.
    3. Utiliza múltiplos tipos de aproximação. Por exemplo, um verificador pode utilizar aproximações diferentes para variáveis diferentes.
    4. Um verificador pode, a princípio, ficar executando por tempo indeterminado a procura de erros.

- Analisadores (*Data Flow Analyzers*)

    1. O objetivo é coletar informação sobre o comportamento dinâmico do programa/especificação.
    2. Um *trace* potencialmente divergente não pode ser simplesmente ignorado. O analisador tem que produzir um resultado que reflita todos os possíveis comportamentos do programa.
    3. Possui um propósito específico (ex.: *pointer analysis* e *strict analysis*). Por conseguinte, em geral, apenas um tipo de aproximação é utilizada.
    4. Um analisador não deve, a princípio, entrar em *loop*.

Figura 5.6: Diferenças entre verificadores e analisadores de código

gerar diferentes instâncias de um mesmo ponto de controle. Esta operação de "juntar" informações permite que o compilador gere assertivas que possibilitarão a aplicação de otimizações (vide o algoritmos das Figura 2.17 e 2.19). Desta maneira, podemos aplicar o operador *join* durante a interpretação, ou após a geração do mapa de estados. A primeira opção é mais eficiente, pois diminui o número de estados gerados durante a análise, entretanto, esta é menos precisa. Nos capítulos posteriores apresentaremos analisadores, de diferentes graus de complexidade implementados em nosso "framework".

5.6.3.1 Analisadores interprocedurais

Em análises interprocedurais é muito mais complexo garantir a terminação da análise. O problema está relacionado a funções [16] recursivas. Por exemplo, a linguagem contendo os comandos *unfolding* e *unfold* (Seção 5.5.1) demonstra este problema. Ou seja, sempre que uma recursão não é de cauda [17] aparecem problemas de terminação (vide exemplo $unfolding(unfold; skip)$). Observe que este não é apenas um problema deste exemplo. Os algoritmos tradicionais de *análise de fluxo de dados* se aplicam somente a análises intraprocedurais, ou análises interprocedurais envolvendo apenas funções onde o único tipo de recursão permitido é o de cauda. A solução apresentada na Seção 5.5.1 para as descrições *SOS* contendo recursão geral é ineficiente e pode ser aplicada apenas a programas muito simples.

Desenvolvemos, então, uma nova abordagem para a realização de análises interprocedurais. A nossa solução é baseada no conceito de *contexto de uso*. Um *contexto de uso* descreve o ambiente de invocação de uma função. Em uma linguagem funcional pura, por exemplo, o contexto de uso é definido pelo valor dos argumentos. Por outro lado, em uma linguagem imperativa, o contexto de uso envolve também o estado da memória. A partir deste conceito, definimos a noção de *efeito de execução*. Dado um *contexto de uso* para uma função, o *efeito de execução* descreve o resultado da execução. Em uma linguagem funcional pura, o *efeito de execução* é definido como o valor retornado pela função. Em um linguagem imperativa o estado da memória após a execução também é considerado. Tendo como base estas duas definições, utilizamos a notação $f[c] \rightarrow e$ para detonarmos que dada uma função f e um contexto c é produzido um efeito e. Para definirmos o analisador interprocedural temos, então, que criar versões abstratas do *contexto de uso* e *efeito de execução*. Temos assim que $f[c_a] \rightarrow e_a$, onde c_a é um contexto abstrato, e e_a um efeito abstrato. Para computarmos o efeito de uma função recursiva f, consideramos inicialmente que o efeito de f é igual ao da função totalmente divergente [18], e iteramos até encontrarmos o ponto fixo. A convergência é garantida em um número finito de passos graças ao uso de aproximações.

Para funções mutuamente recursivas o algoritmo funciona de forma semelhante. Dentro de uma iteração de cálculo do ponto fixo o algoritmo funciona da mesma forma que os algoritmos de análise intraprocedural. Esta abordagem permite que reduzamos um problema de análise interprocedural a uma série de problemas de análise intraprocedural. A Figura 5.7 contém a descrição informal do algoritmo.

A aproximação do *contexto de uso* é determinante para eficiência e precisão do algoritmo. Se assumirmos um contexto de uso vazio, onde todas as invocações a uma função f são consideradas iguais, teremos então um algoritmo eficiente. Porém, extremamente impreciso, porque estaremos considerando milhares de caminhos impossíveis de serem executados, visto que uma função é, em geral, invocada em diferentes partes e contextos de um programa. Esta situação pode ser exemplicada no caso de termos uma função f que é invocada pelas funções $g_1, \ldots, g_n$. Neste

[16] Procedimentos, métodos ou predicados.
[17] *Tail recursive.*
[18] $\forall c \cdot f[c] \rightarrow \bot$.

Quando um função f é invocada em um contexto c, a partir de $g[c']$, temos as seguintes situações:

- Se $f[c]$ já foi analisado. Então, temos que, $f[c] \rightarrow e$, e a análise de $g[c']$ prossegue assumindo e.
- Se $f[c]$ ainda não foi analisado. Então, interrompemos a análise de $g[c']$ e iniciamos a análise de $f[c]$, assumindo a aproximação inicial $f[c] \rightarrow e_0$, onde $e_0 = \perp$.
- Se $f[c]$ ainda não terminou de ser analisada. Este caso reflete uma recursão no programa. Como $f[c]$ ainda não terminou de ser analisada, utilizamos o *efeito de execução* parcial associado a $f[c]$. Em uma primeira iteração este valor é $\perp$. Marcamos $g[c']$ como dependente de $f[c]$.

Ao determinarmos o *efeito de execução* e_n de $f[c]$, temos as seguintes situações:

- Se não existe nenhuma $h[c'']$ dependente de $f[c]$, ou seja, não foi detectado um ciclo envolvendo $f[c]$, então o processo é finalizado, e o analisador armazena o resultado $f[c] \rightarrow e_n$. O analisador continua o processamento de $g[c']$ que invocou $f[c]$, assumindo o *efeito de execução* e_n.
- Se $e_n = e_{n-1}$, onde e_{n-1} é o *efeito de execução* computado na iteração anterior, então o processo é finalizado, e o analisador armazena o resultado $f[c] \rightarrow e_n$. O analisador continua o processamento de $g[c']$ que invocou f, assumindo o *efeito de execução* e_n.
- Se $e_n \neq e_{n-1}$ e existem $g[c']$ dependentes de $f[c]$, ou seja, $f[c]$ está contido em uma recursão e o analisador refinou o *efeito de execução* de $f[c]$. Então, todas as $g[c']$ dependentes de $f[c]$ são marcadas para serem reanalisadas. O analisador passa a assumir $f[c] \rightarrow e_n$.

Figura 5.7: Algoritmo para análise interprocedural

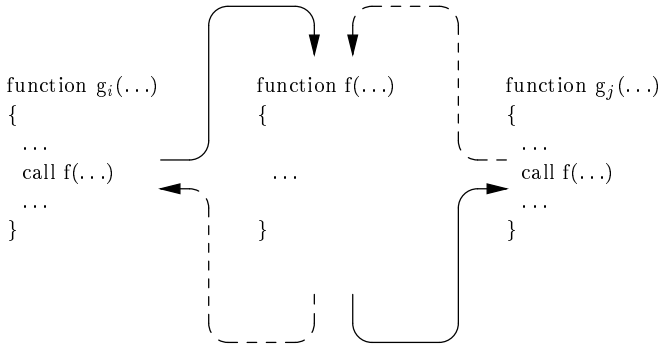

Figura 5.8: Exemplo de caminho impossível de ser executado

exemplo, a informação gerada pela invocação da função f a partir de uma função g_i é propagada para todas as funções $g_1, \ldots, g_n$. É como se estivéssemos considerando caminhos do tipo:

- g_i invoca f;
- f é executada;
- execução retorna para g_j, onde j pode ser diferente de i (Figura 5.8).

A nossa abordagem abstrai diversas soluções propostas para a análise interprocedural. *Sharir* e *Pnueli* descreveram um método de abstração denominado *call strings* [88]. Este método pode ser refletido em nossa abordagem, definindo o *contexto de uso* como as *call strings*. A abordagem *nCFA* [89] de *Shivers* pode ser entendida no escopo da nossa proposta como definindo o *contexto de uso* como as n últimas funções contidas na pilha de execução. Suponha, que a função g_1 invocou g_2, que por sua vez invocou g_3, que em seguida invocou g_4. Considerando $n = 2$, o *contexto de uso* de g_4 é $[g_3, g_2]$.

No capítulo sobre trabalhos correlatos mostramos uma comparação do nosso método de análise de código com a técnicas convencionais de análise de fluxo de dados.

5.7 Usando "Marcações"

Na Seção 5.6.2.5 descrevemos como algumas marcações podem melhorar o resultado da análise de programas/especificações permitindo que mais otimizações sejam aplicadas. Logo, é interessante que toda a *DSL* possua notação para definição de propriedades que sejam relevantes para a análise e verificação de programas.

Podemos fazer uma analogia com a utilização de sistemas de tipos em linguagens de programação. As declarações de tipos nada mais são do que marcações que são utilizadas pelo compilador para realizar verificações que eliminam determinados erros de execução. Estas declarações também permitem que o compilador gere código mais eficiente. As declarações de tipos são informações adicionais ("marcações") fornecidas pelo desenvolvedor com o intuito de minimizar os erros de execução e melhorar a eficiência do código gerado.

O compilador *GNU C++* também possui "marcações" que melhoram os resultados da análise. A "marcação" *const*, por exemplo, indica que um método não pode modificar o valor de nenhum membro do objeto. O compilador gera mensagens de erro se a condição não for satisfeita. Caso a condição seja satisfeita, o compilador utilizará esta informação para melhorar o resultado das análises de fluxo de dados [19].

[19] O compilador da *GNU* realiza apenas análise intraprocedural, quando um método é invocado, a princípio, todos os membros de um determinado objetos podem ter sido modificados. O compilador só assume esta aproximação drástica para métodos que não estão marcados como *const*.

"Marcações" também podem ser vistas como especificações adicionais ou redundantes. No caso das declarações de tipos, o tipo $int \rightarrow string$ especifica que o argumento da função possui a propriedade de ser um número inteiro e que o valor retornado a propriedade de ser uma *string*.

No caso de *DSLs*, as propriedades descritas através de marcações podem ser mais sofisticadas que declarações de tipos, permitindo que análises e verificações mais sofisticadas sejam realizadas.

5.8 Conclusão

Neste capítulo apresentamos o nosso "framework" para a análise e verificação de código, detalhando todas as etapas de construção destas ferramentas. Os analisadores e verificadores construídos com nossa ferramenta são especificados através de uma linguagem lógica denominada *PAN*, que será apresentada no próximo capítulo.

A maior parte dos algoritmos de análise foram descritos, entretanto, os detalhes serão apresentados nos capítulos contendo exemplos de analisadores. Os diferentes tipos de analisadores foram descritos conjuntamente com a "melhor" abordagem de implementação dentro do nosso "framework".

5.8.1 Contribuições

- "Framework" para análise e verificação de código.
- Construção modular de analisadores.
- Reutilização de fragmentos de analisadores de código.
- Solução genérica para analisar descrições *SOS* que não sejam semi-composicionais.
- Técnica de verificação utilizando execução livre + aproximações.
- Técnica "*depth first search* + funções de detecção de divergência" para garantir a terminação da análise.
- Sequências de pares $\langle \mathcal{F}, \mathcal{A} \rangle$ para evitar a utilização "prematura" de aproximações.
- Marcadores *assure* [20] para lidar com as imprecisões geradas pelo uso de aproximações.
- Marcadores *assure* como filtros de *traces*.
- Técnica para análise interprocedural.
- Uso de "marcações" para melhorar os resultados da análise.
- "Marcações" como especificações adicionais ou redundantes.

[20] "Instâncias" da versão simples desta categoria de marcadores pode ser encontrada em algumas linguagens e analisadores de código (ex.: linguagem *Clean*). Mas nunca encontramos na literatura algo parecido com a versão temporal deste marcador.

Capítulo 6

A linguagem *PAN*

6.1 Introdução

Dado uma descrição *SOS*, podemos facilmente construir um interpretador mapeando esta descrição para um programa *Prolog*. Também podemos construir analisadores e verificadores de código em *Prolog* utilizando aproximações. Mas a eficiência de nossos interpretadores, analisadores e verificadores seria muito ruim, e o consumo de memória seria inaceitável. Apesar de existirem inúmeros trabalhos sobre otimização de programas lógicos [4, 83, 42, 23], nenhum deles atendem as nossas exigências. Portanto, decidimos implementar um compilador para a nossa própria linguagem lógica (*PAN*) destinada principalmente para o desenvolvimento de interpretadores, analisadores e verificadores de código. Para atingirmos os nossos objetivos, o nosso compilador tem que realizar diversas análises. Ao contrário da maioria dos compiladores, essas análises não são locais, e sim globais, aumentando, portanto, em muito o tempo de compilação. Entretanto, isso não é um grande problema, porque podemos desabilitar essa fase de análise/otimização durante o desenvolvimento de um analisador e reabilitá-la somente durante a geração da versão final do analisador/interpretador.

A sintaxe/semântica de *PAN* possui algumas similaridades com *Prolog*, mas também existem diversas diferenças. Estamos assumindo que o leitor possua alguma familiaridade com *Prolog* ou outra linguagem lógica. As principais diferenças de *PAN* em relação a *Prolog* são:

- *PAN* possui um sistema de tipos polimórficos semelhante ao da linguagem ML [62].
- *PAN* possui suporte a declarações *mixfix*. *Prolog* só possui suporte a declarações *infix* e *prefix*.
- *PAN* possui suporte a *overloading*. Em *Prolog*, só podemos redefinir um predicado/funtor se estes possuirem aridades diferentes.
- *PAN* possui suporte a módulos. Algumas implementações de *Prolog* também possuem o conceito de Módulo.
- *PAN* possui uma interface "limpa" com a linguagem *C*.
- *PAN* possui uma quantidade muito restrita de recursos não lógicos. Por exemplo, *PAN* não possui predicados como *assert* e *retract* de *Prolog*. Enfim, o único recurso não lógico de *PAN* é o *cut*.

6.2 Estrutura de um Programa *PAN*

Cada arquivo "*.pan*" contém um módulo [1]. A primeira declaração deste arquivo deve ser o nome do módulo. Suponha que estejamos definindo um módulo chamado *simple-language*, neste caso, teríamos algo como:

```
module simple-language.
```

Após a declaração do nome do módulo, devemos declarar quais módulos serão importados. Por exemplo, suponha que o módulo *simple-language* importe os módulos *util*, *std-lib* e *analysis-support*, logo teríamos algo como:

```
import util, std-lib.
import analysis-support.
```

Em seguida, um módulo *PAN* contém as especificações das assinaturas e predicados. Por exemplo, considere a especificação do predicado de concatenação de listas:

```
append nil Xs Xs.
append (X :: Xs) Ys (X :: Zs) :- append Xs Ys Zs.
```

Esta especificação contém quatro variáveis X, Xs, Ys, Zs, uma constante lógica `:-` (representando a implicação invertida), e três constantes não lógicas, denominadas `nil`, `::` e `append`, denotando a lista vazia, o construtor de lista e o predicado de concatenação, respectivamente. Duas dessas constantes, `::` e `:-` são símbolos *infix*. Esta especificação só possui algum significado se existirem declarações de *tipagem* e *mixfix* para as constantes utilizadas. Neste caso, teríamos algo como:

```
type _ :- _ : o -> o -> o {prec 0}.
type nil : list A.
type _ :: _ : A -> list A -> list A {prec 5 l-assoc}.
type append : list A -> list A -> list A -> o.
```

Diversas dessas constantes são *builtin* em *PAN*, neste exemplo apenas o predicado `append` não é *builtin*. Todas as constantes que não forem *builtin* devem possuir uma declaração. Porém no exemplo acima, foram introduzidas duas novas constantes `o` e `list`. Essas constantes também precisam de declarações que serão denominadas de *kind declarations*. Podemos ver estas declarações como os tipos dos tipos. A variável `A` é uma *type variable*, que possui um propósito semelhante as *type variables* existentes na linguagem ML. Ao contrário de linguagens como *Haskell* [49], *PAN* não possui um sistema de inferência de tipos sofisticado. Em *PAN* somente o tipo das variáveis é inferido. O tipo das constantes tem que ser declarado. Esta limitação está relacionado ao suporte a declarações *mixfix* e ao *overloading*, que complicam em muito um sistema de inferência de tipos.

6.3 *Kind Declarations*

Kind declarations são utilizadas para introduzirmos construtores de tipos. Uma *kind declaration* começa com o símbolo *kind* seguido por uma lista de identificadores e uma expressão de *kind*. Por exemplo, a *kind declaration* associada às constantes `o` e `list` seria:

```
kind o : type.
kind list : type -> type.
```

[1] Em futuras versões cada arquivo poderá conter mais de um módulo.

Estas declarações definem `o` como um tipo e `list` como um construtor que recebe um tipo, e retorna um novo tipo. Por exemplo `list int` é o tipo das listas de números inteiros, e `list (list int)` é o tipo das listas de listas de números inteiros. Similarmente, podemos definir a constante:

```
kind pair : type -> type -> type.
```

Esta recebe dois tipos e retorna um novo tipo. Por exemplo,

```
pair int (list int)
```

Esta seria o tipo dos pares formados por um número inteiro e uma lista de números inteiros. Em *PAN* as seguintes *kind declarations* são *builtin*:

```
kind o, int, real, string, qid : type.
kind list : type -> type.
```

A constante `o` possui um significado especial em *PAN*, esta representa o principal tipo da linguagem e pode ser visto como um "booleano".

6.4 *Type Expressions*

A partir dos construtores de tipos, *type variables* e do construtor funcional, podemos construir *type expressions*. Identificadores que comecem com letra maiúscula representam *type variables* e são utilizados para definirmos tipagem polimórfica. O construtor funcional `->` é um construtor *infix*, que é associativo à direita, logo a expressão `a -> b -> c` é analisada sintaticamente como `a -> (b -> c)`. Ao considerarmos as *kind declarations* da seção anterior, temos as seguintes *type expressions*:

```
int -> int -> pair int int
A -> B -> list (pair A B)
list A -> list A -> list A -> o
(A -> B -> o) -> list A -> list B -> o
```

O construtor `->` possui a menor prioridade, logo a expressão `list A -> B` deve ser lida como `(list A) -> B`.

6.5 *Type Declarations*

As *type declarations* começam com o símbolo *type* seguido por um ou mais identificadores *mixfix*. Um identificador simples é formado por uma letra minúscula seguida de letras, números e os símbolos `-` e `'`. Um identificador *mixfix* é formado por identificadores simples, localizadores de argumento (`_`) e os símbolos `+ - * / ^ < > = : , ; ? @ | ! # % \ $ & [ ] \ .` Os localizadores de argumento possuem a função de identificar a posição em que um argumento é esperado. Por exemplo, o identificador *mixfix* `_::_` indica que um argumento é esperado na primeira posição, que por sua vez é seguido pelo símbolo `::` e por mais um argumento. Identificadores *mixfix* permitem a definições muito mais complexas, tais como `if_then_else_` e `[_:_]`.

Após a definição dos identificadores *mixfix*, uma *type declaration* contém uma *type expression* que especifica o tipo das constantes associadas aos identificadores *mixfix*. Opcionalmente, podemos especificar a precedência e a associatividade de uma nova constante. A seguir temos alguns exemplos de *type declarations*:

```
type if_the_else_ : expr -> stmt -> stmt -> stmt.
type _ + _ : expr -> expr -> expr {prec 50 l-assoc}.
type _ := _ : id -> expr -> stmt.
type _ ; _ : stmt -> stmt -> stmt {prec 100 l-assoc}.
type _ :: _ : A -> list A -> list A {prec 5 r-assoc}.
type append : list A -> list A -> list A -> o.
type map : (A -> B -> o) -> list A -> list B -> o.
type is-member : A -> list A -> o.
```

Identificadores *mixfix* que sejam formados apenas por um identificador simples são tratados de forma especial. O compilador assume uma definição implícita dos localizadores de argumento. No exemplo anterior, as declarações de `append`, `map` e `is-member` são processadas como `append _ _ _`, `map _ _ _` e `is-member _ _`.

As *type declarations* cujas *type expressions* terminam com `o` são denominadas de predicados, as demais são consideradas construtores. Por exemplo, `append`, `map` e `is-member` são predicados e as demais são construtores. Os predicados são o análogo em *PAN* de funções e procedimentos existentes em linguagens como *ML* e *C*. Uma analogia mais precisa seria dizer que predicados são procedimentos como suporte a *backtracking*. Os construtores funcionam como criadores de elementos de um determinado tipo, por exemplo, o construtor `_:=_` pega um objeto do tipo `id` e um do tipo `expr` e cria um objeto do tipo `stmt`. Analogamente, o construtor `_::_` pega um objeto de qualquer tipo `A` e um objeto do tipo `list A`, e retorna um objeto do tipo `list A`.

No exemplo anterior, `_+_` possui precedência 50 e é associativo à esquerda, `_;_` possui precedência 100 e é associativo a esquerda, e `_::_` possui precedência 5 e é associativo a direita.

Nem todas as *type declarations* são válidas. Para evitarmos a realização de verificações de tipagem em tempo de execução, o que diminuiria a eficiência do código gerado, existem restrições na *type expression* associada a um construtor. Portanto, exigimos que todas as variáveis que apareçam em uma *type expression* associada a um construtor devam também aparecer no último elemento desta *type expression*. Esta noção de "último" é definida como:

$$last(t) = \begin{cases} last(t_2) & \text{se } t = t_1 \rightarrow t_2 \\ t & \text{caso contrário} \end{cases}$$

Por causa desta restrição, a seguinte *type declaration* é inválida:

```
type _::_ : A -> lst -> lst.
```

Esta declaração permitiria que tivéssemos listas formadas por elementos de diferentes tipos, o que certamente exigiria checagens de tipo em tempo de execução.

Outra restrição é relativa ao *nesting* do operador `->`. Neste caso, permitimos apenas o *nesting* de predicados, logo, as seguintes declarações são válidas:

```
type map : (A -> B -> o) -> list A -> list B -> o.
type tst : (A -> B -> o) -> (B -> A -> o) -> struct A B.
```

Mas as seguintes declarações não são válidas:

```
type mapF : (A -> B) -> list A -> list B -> o.
type tstF : (A -> B) -> (B -> A) -> struct A B.
```

Todo módulo *PAN* permite declarações locais que não são visíveis fora do módulo. As *kind declarations* e *type declarations* locais são definidas através dos símbolos `localkind` e `local` respectivamente.

6.6 Declaração de Variáveis

Apesar de em *PAN* os tipos das variáveis serem inferidos automaticamente, permitimos que o usuário especifique explicitamente os tipos das variáveis. Por exemplo, as declarações:

```
var Xs, Ys : list A.
var L : label.
```

Estas definem que o tipo das variáveis `Xs` e `Ys` é `list A`, e o tipo da variável `L` é `label`. Todas as declarações de variáveis são locais ao módulo.

6.7 Cláusulas

As *kind declarations* e *type declarations* constituem a assinatura de um módulo *PAN*. Mas a assinatura sozinha é de pouca utilidade. Em *PAN*, o programa propriamente dito é formado por uma lista de cláusulas [2] que podem ser vistas como o "corpo" dos predicados (procedimentos).

Cada regra ou cláusula em *PAN* pode ser (simplificadamente) como um comando

$$H\text{:-}B_1, B_2, \ldots, B_n$$

Onde $n \geq 0$. O *goal* H é denominado de cabeça da regra, e a conjunção de *goals* $B_1, B_2, \ldots, B_n$ é denominada de corpo da regra. Quando $n = 0$, chamamos a regra de fato. Por exemplo, a regra:

```
kind person : type.
type _ is grandfather _ : person -> person -> o.
type _ is father _ : person -> person -> o.
X is grandfather Y :- X is father Z, Z is father Y.
```

Pode ser entendida declarativamente como "Para todo `X` e `Y`, `X` é `grandfather` de `Y` se existe `Z`, tal que `X` é `father` de `Z`, e `Z` é `father` de `Y`". Por outro lado, esta regra pode também ser entendida de forma procedural como "Para responder a consulta `X` é `grandfather` de `Y`, responda a consulta conjuntiva `X` é `father` de `Z`, e `Z` é `father` de `Y`".

Em *PAN* também permitimos que as cláusulas sejam definidas através de uma notação alternativa, que inverte as posições do corpo e da cabeça cláusula. Para utilizarmos esta notação, temos que usar a constante `|-` no lugar de `:-`. Esta notação torna direto o mapeamento as regras *SOS* e as regras *PAN*. Por exemplo a regra *SOS*

$$\frac{S_1 \stackrel{\alpha}{\rightarrow} S_1'}{S_1; S_2 \stackrel{\alpha}{\rightarrow} S_1'; S_2}$$

é mapeada na regra *PAN*

```
S1 -- L --> S1'
|-
S1 ; S2 -- L --> S1' ; S2.
```

assumindo as declarações

```
kind ast, label: type.
type _ ; _ : ast -> ast -> ast.
type _ -- _ --> _ : ast -> label -> ast -> o.
```

[2] Isto é, Cláusulas de Horn.

Do ponto de vista semântico não há nenhuma diferença entre `:-` e `|-`, logo devemos considerar `|-` como um *syntax sugar* que facilita o mapeamento de descrições *SOS* para *PAN*.

PAN também permite o uso de disjunções, *if-then-else's* e *not's* dentro do corpo das cláusulas. A semântica dessas constantes é igual a de *Prolog*. A única diferença é que o nosso compilador poder gerar mensagens de "alerta" sempre que este detecta usos inválidos do *if-then-else* e do *not*, i.e. quando eles não condizem com a interpretação lógica [94]. Entretanto, a implementação atual do compilador não gera estas mensagens de alerta. Considere então os seguintes exemplos de cláusulas:

```
type member : A -> list A -> o.
member X Xs :- (Xs = X :: Ys) ; (Xs = Y :: Ys, member X Xs).

type non-member : A -> list A -> o.
non-member X (Y :: Ys) :- not X = Y, non-member X Ys.
non-member X nil.

type add-element : A -> list A -> list A -> o.
add-element E Xs Ys :-
      if (member E Xs)
      then (Ys = Xs)
      else (Ys = E :: Xs).
```

A sintaxe das cláusulas é bastante semelhante a de *Prolog*, a grande diferença é a notação *mixfix*. Como podemos ver nestes exemplos, a sintaxe de *PAN* é extremamente ambígua, tornando impossível analisá-la utilizando-se um analisador sintático LaLR. Os interessados devem consultar o Apêndice B para entenderem o funcionamento do analisador sintático da linguagem *PAN*.

6.8 Interface com *C*

O compilador *PAN* transforma um programa *PAN* em um programa *C*, permitindo que código *PAN* chame código *C*, e vice-versa. Este recurso é fundamental para a implementação dos nossos interpretadores e analisadores, porque o código responsável pela construção do mapa de estados (sistema de transição) é todo codificado em *C*.

Em um programa *PAN*, podemos instruir o compilador a inserir fragmentos adicionais de código *C* no programa gerado. Para isso, basta usarmos uma declaração semelhante a:

```
C{{
// código C
...
}}C.
```

Esta declaração é particularmente útil para incluirmos arquivos ".h" adicionais no programa gerado. Por exemplo, para incluirmos o arquivo "stdio.h", devemos colocar a seguinte declaração no programa: *PAN*.

```
C{{
#include<stdio.h>
}}C.
```

Um programa *PAN* pode possuir um ou mais pontos de entrada que possam ser invocados a partir de código *C*. Mas em linguagens lógicas não há a noção de valor de entrada e retorno,

já que um predicado pode ser utilizado de diversos modos. Por exemplo, o predicado *append* pode ser utilizado com os dois primeiros parâmetros instanciados, e o terceiro parâmetro livre, neste caso, este pode ser visto como uma função que concatena os dois primeiros parâmetros e retorna o resultado no terceiro parâmetro. Por outro lado, este também pode ser utilizado com os dois primeiros parâmetros livres e o terceiro instanciado, neste caso, este pode ser visto como uma função que gera todas as possíveis listas que concatenadas resultam no terceiro parâmetro. Quando interfaceamos com a linguagem *C*, o modo de uso tem que estar explícito, para o código *C* "saber" o que é parâmetro de entrada e o que é de saída. Então, um predicado que seja um ponto de entrada deve anotar quais parâmetros são de entrada e quais são de saída. No seguinte exemplo temos:

```
entry run : in program -> out store -> o.
```

Este predicado indica que o primeiro parâmetro, que pertence ao tipo `program`, é de entrada. Um parâmetro é dito de entrada, se este está totalmente instanciado. O segundo parâmetro, que pertence ao tipo `store`, é de saída. Um parâmetro é dito de saída, se este está completamente não instanciado, i.e. está livre. Esta notação não permite que definamos predicados com parâmetros parcialmente instanciados. Porém, isso não é problema, visto que sempre é possível definir um novo predicado que respeite as nossas limitações. Suponha que desejamos invocar o predicado `tst` com o primeiro parâmetro sendo uma lista com a cabeça instanciada e a cauda não instanciada, e o segundo parâmetro totalmente livre. Logo, podemos definir o predicado `tst2` com três parâmetros da seguinte forma:

```
type tst : list A -> list A -> o.

entry tst2: in A -> out list A -> out list A -> o {c-name test}.
tst2 X Xs Ys :- tst (X :: Xs) Ys.
```

Numa versão futura do compilador, iremos realizar esta transformação automaticamente. A marcação `c-name test` indica o nome que será utilizado no programa *C* para invocar esse predicado. Este tipo de marcação também é utilizada para construirmos objetos *PAN* em *C*. Por exemplo, considere o seguinte exemplo:

```
kind stmt : type.
type _ ; _ : stmt -> stmt -> stmt {c-name seq}.
```

Neste caso, o compilador *PAN* irá criar uma função *C* chamada `seq` que poderá ser utilizada para criar objetos do tipo especificado.

Predicados também podem ser implementados por fragmentos de código *C*, permitindo que código *PAN* chame código *C*. Mas uma vez, tem que estar explícito quais parâmetros são de entrada e quais parâmetros são de saída. O usuário também poderá utilizar marcações para indicar se o predicado é determinístico ou não, permitindo que o compilador gere código eficiente. Considere o seguinte fragmento de código:

```
cdecl tst : in stmt -> in stmt -> out stmt -> o {det}
C{{
// código C
...
}}C.
```

Este fragmento de código define o predicado determinístico `tst` que recebe dois parâmetros de entrada e um de saída.

Além das marcações `in` e `out`, o compilador também possui marcações que indicam se um parâmetro pode ser atualizado destrutivamente. Este recurso é fundamental em nosso trabalho, visto que desejamos atualizar destrutivamente o mapa de estados, e ao mesmo tempo, não queremos perder a interpretação declarativa do programa. Linguagens funcionais puras possuem o mesmo problema. Estas não possuem atualizações destrutivas, logo, a princípio, toda vez que um *array* for atualizado uma nova cópia do *array* terá que ser criada. Analogamente, no caso dos nossos analisadores, toda a vez que o mapa de estados tivesse que ser atualizado, uma nova cópia do mapa de estados deveria ser criada. Se levarmos em conta que o mapa de estados é uma estrutura de dados enorme, isso implicaria em uma grande perda de eficiência. Para resolvermos este problema, sem perder a interpretação declarativa do programa, utilizamos então uma solução semelhante a encontrada em algumas linguagens funcionais puras [101, 77]. Nesta solução, são utilizadas marcações que indicam que o parâmetro deve possuir uma propriedade que garanta que é o seguro a realização de atualizações destrutivas. Se ignorarmos a motivação teórica [3], o parâmetro deve possuir a propriedade que só existe um ponteiro apontando para ele, isto é, não existem duas variáveis apontando para (referenciando) o mesmo objeto. Nesta situação, a atualização destrutiva pode ser feita de forma segura. Em nosso compilador utilizamos a marcação `unq` para designarmos esta propriedade. Obviamente, a atualização destrutiva apenas é permitida em contextos onde o valor atualizado não será necessário em um *backtracking*. Apesar de o analisador sintático reconhecer estas marcações, a implementação atual não verifica se predicados contendo marcas `unq` são corretamente invocados. Suponha que desejamos implementar em *C* suporte a *arrays*. Neste caso, deveríamos utilizar o seguinte trecho de código *PAN*:

```
kind array : type -> type.

cdecl create-empty-array : unq out array A -> o {det}
C{{
// código C
...
}}C.

cdecl update-array : in A -> unq in array A ->
                     unq out array A -> o {det}
C{{
// código C
...
}}C.

cdecl get : unq in array A -> out A -> o {det}
C{{
// código C
...
}}C.
```

Com base nestas definições, o seguinte trecho de código seria um programa *PAN* válido:

```
..., create-empty-array A0,
     update-array 10 A0 A1,
     update-array 20 A1 A2, get V A2,
...
```

[3] A motivação teórica é baseada em lógica linear, os interessados devem consultar [101].

Entretanto, o seguinte trecho de código não é válido

```
..., create-empty-array A0,
     A1 = A0,
     update-array 10 A0 A2,
...
```

Porque `A0` e `A1` "apontam" para o mesmo objeto, assim se atualizarmos este objeto destrutivamente e fizermos `A2` apontar para ele, teremos um problema, já que `A1` deveria estar "apontando" para um *array* vazio, e não um contendo o número 10.

6.9 Predicados de Alta Ordem

PAN possui suporte a predicados de alta ordem e ao *curring*. Este recursos seriam equivalentes aos existentes em linguagem funcionais como *ML* e *Haskell*. Em resumo, *PAN* permite a *passagem* de predicados como argumentos de outros predicados. Por exemplo, podemos definir o predicado:

```
type map : (A -> B -> o) -> list A -> list B -> o.
map F nil nil.
map F (X :: Xs) (Y :: Ys) :- (F X Y), map F Xs Ys.
```

O predicado `map` recebe um predicado como parâmetro que será "aplicado" aos elementos da lista. Em outras palavras, o predicado `map` garante que os elementos de cada lista são relacionados pelo predicado recebido como parâmetro. Em *PAN*, somente predicados *prefix* podem ser passados como parâmetros de predicados de alta ordem.

De forma semelhante as linguagens funcionais, podemos realizar o *currying*, que nada mais é do que "fixar" os argumentos de um predicado. Por exemplo, podemos utilizar o predicado `map` da seguinte maneira:

```
type add : int -> int -> int -> o.
add X Y Z :- Z is X + Y.

type add10-to-members : list int -> list int -> o.
add10-to-members Xs Ys :- map (add 10) Xs Ys.
```

Neste exemplo, o predicado `add10-to-members` adiciona 10 a cada um dos membros da lista `Xs`. O argumento `(add 10)` de `map` é um *curring* que está fixando o primeiro parâmetro do predicado `add`.

6.10 Conclusão

A linguagem *PAN* possui diversos recursos que são fundamentais para a implementação de nossos interpretadores, analisadores e verificadores. A notação *mixfix* provê grande flexibilidade na definição da árvore de sintaxe abstrata e na semântica da linguagem a ser analisada. A interface com a linguagem *C* permite a "comunicação" com o mundo "exterior", como por exemplo, com o analisador sintático da linguagem a ser analisada. O suporte as atualizações destrutivas garantem uma eficiência satisfatória quando estamos lidando com estruturas enormes como os mapas de estado.

6.10.1 Contribuições

- Definição de uma nova linguagem lógica, com os seguintes recursos não presentes em *Prolog*:

  1. Sistema de tipos polimórfico;
  2. Notação mixfix;
  3. Módulos;
  4. Interface limpa com a linguagem *C*;
  5. Atualizações destrutivas que não interferem na interpretação declarativa do programa;
  6. *Curring*.

Capítulo 7

Implementação de *PAN*

7.1 Introdução

O compilador *PAN* realiza diversas análises e transformações no código com o intuito de gerar código eficiente. Os prinpais módulos do compilador estão descritos na Figura 7.1. Este capítulo possui dois propósitos: o primeiro é apresentar detalhes da implementação do compilador; o outro é apresentar diversos analisadores que foram construídos utilizando as técnicas descritas nos capítulos anteriores. Afim de não nos alongarmos muito neste capítulo, evitamos mostrar detalhes relativos a técnicas conhecidas de compilação, tais como: otimização de recursão, otimização de alocação de registradores via coloração de grafos, etc.

A sintaxe de *PAN* é extremamente ambígua, tornando impossível analisá-la utilizando-se um analisador sintático *LaLR*. Os interessados devem consultar o Apêndice B para entenderem o funcionamento do analisador sintático da linguagem *PAN*. Em resumo, utilizamos um analisador sintático *LaLR* para transformar o código em um formato que denominamos pré árvore de sintaxe abstrata. A função do analisador sintático *LaLR* é processar as partes simples do programa. No caso de *PAN*, todas as declarações podem ser processadas facilmente por um analisador sintático *LaLR*. As cláusulas não podem ser processadas. O analisador sintático *LaLR* se limita a tratar os parênteses que aparecem nas cláusulas, e a transformar os elementos de uma cláusula em *tokens*. A checagem de tipos é realizada em conjunto com a conversão da pré árvore de sintaxe abstrata para o formato de árvore de sintaxe abstrata. O módulo de checagem de tipos utiliza um algoritmo contendo um procedimento de unificação [62] que permite que os tipos das variáveis lógicas sejam inferidos.

A partir da árvore de sintaxe abstrata, realizamos uma série de análises que transformam o código sucessivamente até que seja possível realizar uma conversão eficiente para um programa *C*. Para entendermos a motivação destas análises, temos que explorar sucintamente os mecanismos de execução de programas lógicos.

7.1.1 Variáveis lógicas & Unificação

A principal diferença no uso de estruturas de dados entre linguagens lógicas, como *PAN* e *Prolog*, e linguagens imperativas está na natureza das variáveis lógicas. Variáveis lógicas referenciam indivíduos e não posições de memória. Após uma variável ter referenciado um indivíduo em particular, o valor desta variável não pode ser modificado.

A manipulação de dados em programas lógicos é obtida via o algoritmo de unificação. O algoritmo de unificação garante:

- *Single Assignment*;
- Passagem de parâmetros;

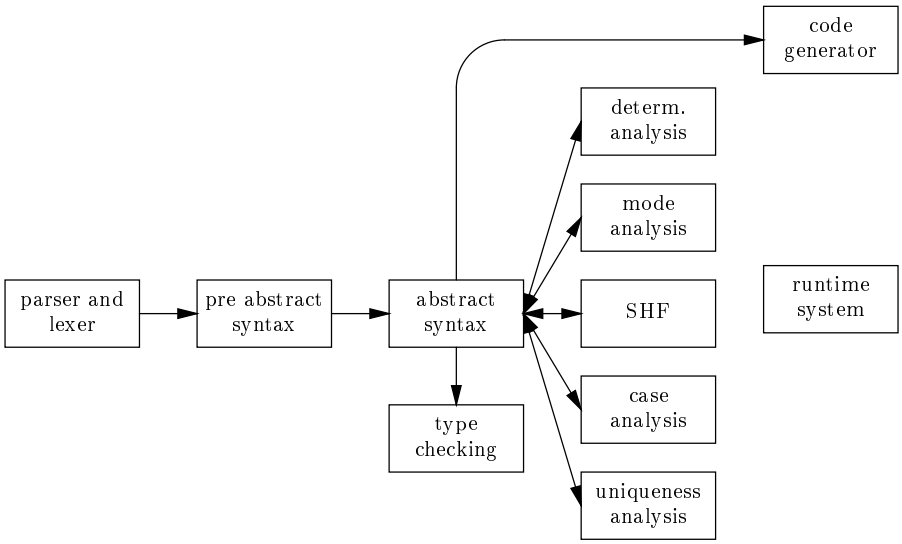

Figura 7.1: Módulos do compilador *PAN*

- Alocação de indivíduos (i.e. estrutura de dados);
- O acesso aos campos de uma estrutura de dados é do tipo *read-many/write-once*.

Portanto, o algoritmo de unificação é o coração do modelo computacional de linguagens lógicas. Um unificador (*unifier*) de dois termos é uma substituição que torna os termos idênticos gerando uma instância comum. Se dois termos possuem um unificador, dizemos que eles são unificáveis. Por exemplo,

```
append (1 :: 2 :: nil) (3 :: nil) Res
```

e

```
append (X :: Xs) Ys (X :: Zs)
```

são unificáveis, porque a substituição [`X = 1`, `Xs = 2 :: nil`, `Ys = 3 :: nil`, `Res = 1 :: Zs`] os torna idênticos, gerando a instância comum:

```
append (1 :: 2 :: nil) (3 :: nil) (1 :: Zs)
```

Um termo s é mais geral que um termo t se existe uma substituição θ tal que $t = s\theta$. O *unificador mais geral*(*mgu*) é o unificador que gera a instância comum mais geral. A Figura 7.2 contém um algoritmo que computa o unificador mais geral(*mgu*). O algoritmo possui chamadas a função *occurs* para evitar a unificação de termos do tipo $f(X)$ e X. Entretanto, a maioria das implementações de *Prolog* omite esta verificação por razões de eficiência. Em *PAN* também omitimos esta verificação.

Este algoritmo de unificação é extremamente ineficiente devido as chamadas a função *substitute*. A maioria das linguagens lógicas (*PAN* inclusive) evita estas substituições. Em suma, as variáveis lógicas e os termos são representados por células de memória. É importante ressaltar que as variáveis livres são, em geral, representadas por células de memória contendo referências para elas mesmas. Ao considerarmos esta representação, podemos substituir a seqüência de comandos:

substitute(s, Y, X); // substitute Y for X in the stack s
substitute(θ, Y, X); // substitute Y for X in θ
add(θ, $\langle X, Y \rangle$); // add the equation $X = Y$ to the substitution θ

```
substitution unify(term T1, term T2)
{
  substitution θ;
  stack s;
  θ = createEmptySubstitution();
  push(s, ⟨T1, T2⟩); // put the equation T1 = T2 in the stack
  while (not isEmpty(s))
  {
    ⟨X, Y⟩ = pop(s); // remove the equation X = Y
                     // from the top of the stack
    if (isVariable(X) and (not occurs(X,Y)))
    {
      substitute(s, Y, X); // substitute Y for X in the stack s
      substitute(θ, Y, X); // substitute Y for X in θ
      add(θ, ⟨X, Y⟩); // add the equation X = Y to the substitution θ
    }
    else if (isVariable(Y) and (not occurs(Y,X)))
    {
      substitute(s, X, Y); // substitute X for Y in the stack s
      substitute(θ, X, Y); // substitute X for Y in θ
      add(θ, ⟨Y, X⟩); // add the equation Y = X to the substitution θ
    }
    else if (areIdenticalConstantsOrVariables(X,Y))
    { continue; }
    else if (X = f(X1, ..., Xn) and Y = f(Y1, ..., Yn))
    {
      for i = 1 to n
          push(s, ⟨Xi, Yi⟩);
    }
    else return failureSubstitution; // terms do not unify
  }
return θ;
}
```

Figura 7.2: Algoritmo de unificação

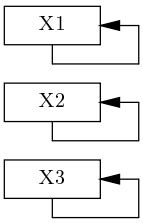

Figura 7.3: Estado inicial da memória

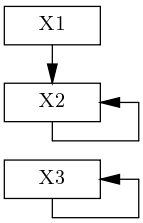

Figura 7.4: Estado da memória, após a primeira unificação

por:

ref(X,Y); //make X a reference to Y

Para exemplificar esta adaptação do algoritmo de unificação, considere o seguinte trecho de código:

```
X1 = X2, X1 = 5, X3 = X1.
```

Considere também, que antes da execução deste trecho de código as variáveis `X1`, `X2` e `X3` estão livres. Antes da execução deste trecho de código, a memória pode ser representada conforme descrito na Figura 7.3. Após a execução de `X1 = X2`, a memória pode ser representada conforme descrito na Figura 7.4. Após a execução de `X1 = 5`, temos a Figura 7.5. Neste caso, fica claro que para acessarmos o valor da variável `X1`, temos que percorrer uma cadeia de ponteiros composta por 2 elementos. Esta operação é denominada de *dereferenciação*. Ao executarmos `X3 = X1` temos a Figura 7.6.

Apesar desta otimização, ainda não é possível comparar a eficiência de um programa lógico com a de um imperativo. Visto que, os processos de passagem de parâmetros, atribuição de valor e alocação de memória são todos realizados pela função de unificação. Entretanto, se conseguirmos identificar subproblemas simples do processo de unificação, poderemos substituir o algoritmo genérico de unificação por instâncias mais simples. Para identificarmos estes subproblemas do processo de unificação, temos que construir um analisador que dado um programa lógico, este infere a instanciação das variáveis. O analisador identifica em cada localização do programa que variáveis estão livres (*free*), parcialmente instanciadas e completamente instanciadas (*ground*). Com posse desta informação sobre a instanciação das variáveis, podemos definir os sequintes subproblemas do processo de unificação:

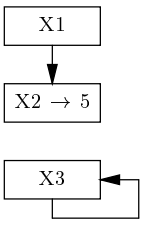

Figura 7.5: Estado da memória, após a segunda unificação

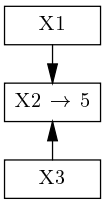

Figura 7.6: Estado da memória, após a terceira unificação

```
append nil X X.
append (X :: Xs) Ys (X :: Zs) :- append Xs Ys Zs.
```

Figura 7.7: Predicado *append*

- $X = Y$, onde X é *free* e Y é *ground*. Este caso será denominado de unificação de atribuição e será representado por $X := Y$. Após a unificação X e Y são *ground*.
- $X = Y$, onde X é *ground* e Y é *ground*. Se X e Y são valores atômicos, esta unificação se resume a um teste, que será representado por $X == Y$. Se o valor de X e Y não forem atômicos (ex.: forem listas), geramos então uma função recursiva que realiza o teste de igualdade, para o tipo associado as variáveis X e Y.
- $X = f(Y_1, \ldots, Y_n)$, onde X é *free*. Neste caso a unificação está funcionando como um "alocador" de memória que será representada como $X\ != f(Y_1, \ldots Y_n)$.
- $X = f(Y_1, \ldots, Y_n)$, onde X é *ground* e os Y_i são *free*. Neste caso, a unificação está funcionando como mecanismo de acesso aos campos (ou "deconstrução") de uma estrutura de dados. Este tipo de unificação será representada como $X\ ?= f(Y_1, \ldots, Y_n)$.

7.2 Conversão para o formato homogêneo

Para simplificar a construção dos analisadores, todo o programa *PAN* é convertido para um formato simplificado, denominado na literatura de *superhomogeneous form* [91] ou *coreProlog* [82]. Obviamente, o nosso formato não é exatamente igual ao definido nestes trabalhos, afinal *PAN* não é exatamente igual a *Prolog* ou *Mercury*[1].

As cláusulas são convertidas para um formato, onde um átomo possui uma das seguintes formas:

- $p(X_1, \ldots, X_n)$ e todos os X_i são variáveis distintas
- $X = Y$ ou $X\ is\ Y$
- $X = f(X_1, \ldots, X_n)$ ou $X\ is\ f(X_1, \ldots, X_n)$ e todos os X_i são variáveis distintas.

Além disso, predicados definidos por mais de uma cláusula são transformados em equivalentes compostos por apenas uma cláusula. Por exemplo, o predicado *append* definido na Figura 7.7 é transformado no predicado contido na Figura 7.8. Note que todas as unificações tornaram-se explícitas.

[1] *Mercury* também é uma linguagem lógica.

```
append V0 V1 V2 :-
   (V0 = nil,
    V2 = V1
   )
   ;
   (V0 = (V3 :: V4),
    V5 = V3,
    V2 = (V5 :: V6),
    V7 = V4,
    V8 = V1,
    V9 = V6,
    (append V7 V8 V9)
   )
```

Figura 7.8: Predicado *append* em *superhomogeneous form*

7.3 *Mode Analysis*

Como foi dito anteriormente, a unificação possui diversos subproblemas simples. Porém para identificarmos estes problemas, temos que construir um analisador que dado um programa lógico, este infere a instanciação das variáveis. Denominaremos de *modo* de um predicado a instanciação dos argumentos do predicado antes e depois da "execução" do mesmo. A definição de *modo* é extremamente útil, já que cada predicado pode ser utilizado de diferentes "modos". Utilizaremos a notação $pre \rightarrow pos$ para designar a instanciação de um argumento antes (pre) e depois (pos) da execução do predicado. O predicado *append* possui os seguintes *modos*:

- $append(ground \rightarrow ground, ground \rightarrow ground, free \rightarrow ground)$, neste *modo* o predicado é determinístico, e "funciona" concatenando as listas referenciadas pelos dois primeiros argumentos e retornando o resultado no terceiro argumento.
- $append(ground \rightarrow ground, ground \rightarrow ground, ground \rightarrow ground)$, neste *modo* o predicado também é determinístico, e "funciona" como um teste que verifica se as duas primeiras listas concatenadas resultam na terceira lista.
- $append(free \rightarrow ground, free \rightarrow ground, ground \rightarrow ground)$, neste *modo* o predicado é não determinístico, e "funciona" gerando todos os pares de listas que concatenadas resultam na lista referenciada pelo terceiro argumento.

A partir deste exemplo fica claro que é vantajoso compilar um mesmo predicado para cada diferente *modo* de uso. Ou seja, o compilador transforma cada predicado em um conjunto de funções, onde cada função corresponde a um *modo* de uso. Esta abordagem é semelhante a utilizada no analisador de ponteiros para linguagem *C* descrito em [103]. Este analisador de ponteiros interprocedural utiliza informação contextual para melhorar o resultado da análise. Portanto, uma função do programa *C* pode ser replicada no código gerado, onde cada réplica está associada a uma informação contextual diferente. No nosso caso, a informação contextual é a instanciação dos argumentos do predicado. O *modo* de um predicado nada mais é do que a definição de um contexto de utilização do predicado. Este processo de gerar diferentes versões de um mesmo predicado pode também ser comparado ao processo de avaliação parcial [51] [2].

[2] Esta não é uma comparação "precisa", queremos apenas apontar as analogias existentes entre os dois processos.

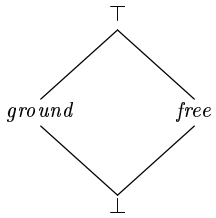

Figura 7.9: Reticulado "simples" para análise de modo

Utilizando informação sobre os argumentos de entrada, podemos especializar um predicado, através da especialização do algoritmo de unificação para cada uma das unificações existentes dentro do predicado.

A príncipio, o nosso analisador de "modos" pode utilizar um reticulado de valores abstratos como o descrito na Figura 7.9. Entretanto, este reticulado produzirá resultados ruins. Em linguagens lógicas é muito comum termos argumentos parcialmente instanciados. Um argumento está parcialmente instanciado, quando um pedaço está instanciado, enquanto que outro pedaço permanece livre. Por exemplo, nos exemplos de *SOS* modular (Capítulo 4), o *label* é um argumento parcialmente instanciado. Nestes casos, um analisador utilizando o reticulado descrito na Figura 7.9 tem que se limitar a responder $\top$ ("não sei").

Por este motivo, devemos utilizar um reticulado mais "rico" para atingirmos os nossos objetivos. Portanto, decidimos utilizar um reticulado semelhante ao definido pelo analisador de tipos [3] descrito em [15]. Este analisador de tipos define uma estrutura de dados denominada de grafo de tipos. No caso do nosso analisador, denominaremos esta estrutura de grafo de *modo*. Em nosso analisador, a função básica deste grafo é representar as estruturas parcialmente instanciadas.

Antes de apresentarmos a definição do grafo de *modo*, é importante notar que a operação de *dereferenciação* é muito custosa. Antes de realizarmos uma unificação, temos sempre que verificar se precisamos ou não realizar esta operação, e conforme o caso realizá-la. Logo, é interessante que o nosso analisador identifique os casos onde não é necessário realizar a operação de *dereferenciação*. Outro problema está relacionado a inicialização das variáveis livres. Uma variável livre é inicializada fazendo-a referenciar a si mesma. Na grande maioria dos casos, a variável livre é inicializada, mas logo em seguida é realizada uma unificação que modifica o valor da variável, tornando, desta maneira, o processo de inicialização um desperdício de "tempo". Portanto, é interessante que o nosso analisador também identifique os casos onde é desnecessário inicializar uma variável. Para exemplificarmos esta situação, considere o fragmento de código a seguir:

```
add5 X Y :- X1 = 5, Y is X + X1.
```

Neste caso, não faz sentido inicializar a variável `X1`, esta é imediatamente unificada como o valor 5. Então, além dos valores abstratos "atômicos" *free* e *ground*, temos:

ground-deref representa um termo completamente instânciado, onde *não* precisa ser realizada a operação de *dereferenciação* durante uma unificação.

uninit representa uma variável *não* inicializada.

deref representa uma variável não inicializada ou um termo completamente instânciado, onde *não* precisa ser realizada a operação de *dereferenciação*. Ou seja, *deref* = *ground-deref* $\sqcup$ *uninit*.

[3] Como *PAN* é uma linguagem tipada, não é necessário implementar um analisador de tipos.

Um grafo de *modo* $G \in \mathcal{G}$ é um grafo finito bipartido, que consiste de:

1. Um conjunto finito N_m de nodos de *modo* (representados como $\circ$ nos diagramas).
2. Um conjunto finito N_f de nodos funtores n_f, rotulados pelos funtores do programa e um *flag* indicando se o funtor precisa ser dereferenciado ou não, e $N_m \cap N_f = \emptyset$. A função *label* retorna o label associado a um dado nodo. A função *is_deref* retorna *true* se o funtor não precisar ser dereferenciado. O conjunto N_f também possui os seguintes valores abstratos atômicos: $\bot$, *ground-deref*, *ground*, *uninit*, *free*, *deref* e $\top$.
3. Uma raiz $r \in N_m$ tal que existe um caminho a partir de r para qualquer nodo de G.
4. Um conjunto $A \in \wp(N_m \times N_f) \cup \wp(N_m \times \mathbb{N} \times N_f)$ de arcos, tal que:
   - Para todo nodo $n_m \in N_m$ existe um arco $a = \langle n_m, n_f \rangle$
   - Para todo nodo $n_m \in N_m$ e para todos arcos $a_1 = \langle n_m, n_{f_1} \rangle$ e $a_2 = \langle n_m, n_{f_2} \rangle$, temos que $label(n_{f_1}) \neq label(n_{f_2})$
   - Para todo nodo $n_m \in N_m$, se existe um arco $a = \langle n_m, atomic \rangle$, então a é o único arco saindo de n_m. Onde *atomic* representa um dos valores abstratos atômicos (*impossible*, *ground-deref*, *ground*, *uninit*, *free*, *deref* e *any*). Temos, também, que $label(atomic) = atomic$.
   - Para todo nodo $n_f \in N_f$, se $label(n_f) = f$, e $f \in \mathcal{F}^n$ com aridade n, então temos n arcos $\langle n_f, i, n_m^i \rangle$, $1 \leq i \leq n$

Utilizamos a notação $n_m : f(n_m^1, \ldots, n_m^n)$ para representar a fórmula

$$\exists n_f \in N_f \cdot \begin{cases} \langle n_m, n_f \rangle \in A \ \wedge \\ label(n_f) = f \in \mathcal{F}^n \ \wedge \\ \forall i \in [1, n] \cdot \langle n_f, i, n_m^i \rangle \end{cases}$$

Um grafo está representando diferentes ou até mesmo infinitas instanciações de variável. A instanciação de uma variável pode ser vista como uma árvore. Dizemos que uma árvores de instanciação t é representada por um grafo com raiz n_m, se:

1. $t = a$, onde *a é valor abstrato atômico* e $n_m : a$
2. $t = c \in \mathcal{F}^0$ e $n_m : c$
3. $t = g(t_1, \ldots t_n)$ e $n_m : g(n_m^1, \ldots, n_m^n)$ e para cada t_i, $1 \leq i \leq n$, temos que t_i é representado pelo grafo com raiz n_m^i.

Podemos, então, definir uma função γ_{tree} que dado um grafo, retorne as árvores de instanciação correspondentes:

$$\gamma_{tree}(G) = \{tree \mid tree \text{ é representada por } G\}$$

Para o grafo G descrito na Figura 7.10, temos que

$$\gamma_{tree}(G) = \{nil, cons(uninit, nil), cons(uninit, cons(uninit, nil)), \ldots\}$$

O diagrama de *Hasse* da Figura 7.11 mostra a relação de ordem entre os valores abstratos atômicos. Podemos definir uma relação "*é aproximada por*" entre instanciações de variáveis e grafos da seguinte forma:

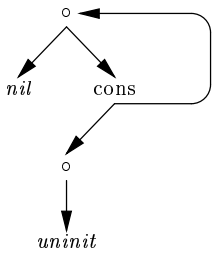

Figura 7.10: Exemplo de grafo de *modo*

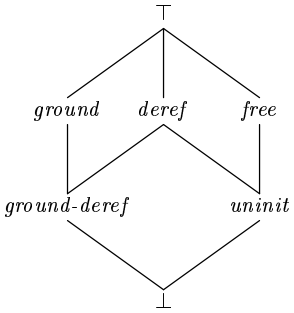

Figura 7.11: Diagrama de *Hasse* para os valores abstratos atômicos

- Uma variável completamente instanciada é aproximada pelos grafos contendo apenas *ground* ou *any*. Se a variável não precisa ser dereferenciada, então, esta também é aproximada pelos grafos contendo apenas *ground-deref* ou *deref*.
- Uma variável livre é aproximada pelos grafos contendo apenas *free* e *any*. Se a variável não tiver *aliases*, então esta também é aproximada pelos grafos contendo apenas *uninit* e *deref*.
- Uma variável parcialmente instanciada é aproximada por um grafo que represente uma árvore que a aproxime

É importante observar que variáveis completamente instanciadas também podem ser aproximadas por grafos que representem árvores completamente instanciadas.

Podemos então definir uma função reificação γ que associe a um grafo todas as instanciações de variáveis correspondentes.

$$\gamma(G) = \{i \mid i \textit{ é aproximada por } G\}$$

A partir da função de reificação podemos definir uma noção de equivalência entre grafos de modo como:

$$G \equiv G' \Leftrightarrow \gamma(G) = \gamma(G')$$

A noção de ordem parcial pode ser definida como:

$$G \sqsubseteq G' \Leftrightarrow \gamma(G) \subseteq \gamma(G')$$

O *poset* de grafos de *modo* não possuem altura finita. A Figura 7.12 mostra um exemplo de seqüência infinita. Através da função de reificação, podemos relacionar esta seqüência crescente e infinita de grafos de *modo* com a seqüência de conjuntos decrita na Figura 7.13.

Devido a existência de seqüências crescentes e infinitas, para garantirmos a convergência do nosso analisador de *modos*, temos que definir um operador de *widening* ($G_1 \bigtriangledown G_2$). Salientamos que o operador *join* ($G_1 \sqcup G_2$) pode ser implementado facilmente através de um algoritmo que

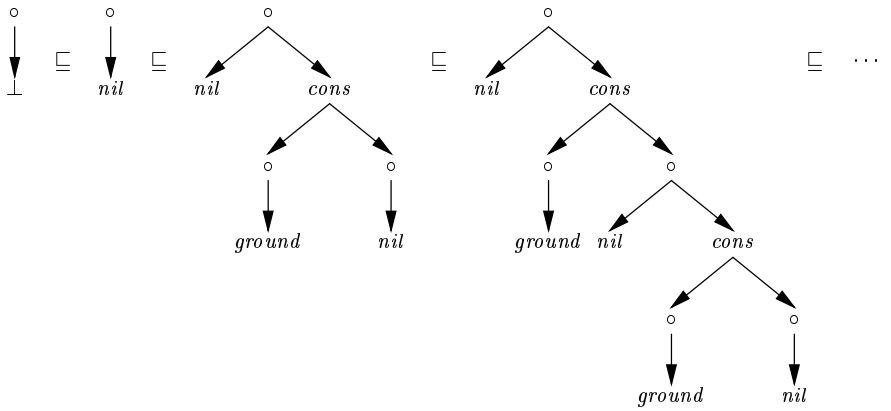

Figura 7.12: Exemplo de seqüencia infinita

$\emptyset \subseteq \{nil\} \subseteq \{nil, cons(ground, nil)\} \subseteq$
$\{nil, cons(ground, nil), cons(ground, cons(ground, nil))\} \subseteq \ldots$

Figura 7.13: Seqüência infinita de conjuntos crescentes

"funde" os nodos de *modo* dos grafos em questão (G_1 e G_2). Por conseguinte, é interessante definir o operador de *widening* utilizando um operador de *partition* ($\odot G$ - Seção 2.6.3). O operador de *partition* do nosso analisador aproxima um grafo de *modo* por outro que em cada caminho acíclico a partir da raiz do grafo, cada funtor aparece no máximo uma vez. Como o número de funtores de um programa lógico é finitio, temos uma quantidade finita de aproximações. Portanto, este operador de *partition* garante a terminação da análise, e o operador de *widening* é definido como $G_1 \bigtriangledown G_2 = \odot(G_1 \sqcup G_2)$.

O operador de *partition* ($\odot G$)pode ser implementado através da aplicação sucessiva das seguintes transformações:

- nodos de *modo* $n_{m_1} : f(n^1_{m_1}, \ldots, n^n_{m_1})$ e $n_{m_2} : f(n^1_{m_2}, \ldots, n^n_{m_2})$ com mesmo funtor f em um caminho acíclico a partir da raiz de G são "fundidos" sem remover-se nenhuma arco.
- nodos de *modo* $n_m \in G$ com filhos distintos $n_m : f(n^1_{m_1}, \ldots, n^n_{m_1})$ e $n_m : f(n^1_{m_2}, \ldots, n^n_{m_2})$, e com o mesmo funtor f terão seus filhos $n^i_{m_1}$ e $n^i_{m_2}$ "fundidos".
- para todo nodos de *modo* n_m que possuam mais de um nodo funtor $(n_{f_1} \ldots n_{f_n})$, e um destes nodos funtores é um valor atômico abstrato. Então devemos substituir todos os nodos funtores de n_m por $n_{f_1} \sqcup \ldots \sqcup n_{f_n}$. [4]

Definindo como contexto de uso de um predicado a instanciação das variáveis antes da execução do mesmo, e como efeito da execução a instanciação das variáveis após a execução, podemos utilizar o algoritmo de análise interprocedural descrito na Seção 5.6.3.1. Contudo, estamos utilizando o operador de *widening* para garantir a convergência do analisador. Considerando, novamente, o predicado *append* com as duas primeiras variáveis instanciadas e a terceira livre e sem *aliases*. O analisador irá gerar a seguinte seqüencia de iterações:

[4] A função desta transformação é garantir que o grafo resultante possui a propriedade "para todo nodo $n_m \in N_m$, se existe um arco $a = \langle n_m, atomic \rangle$, então a é o único arco saindo de n_m. Onde *atomic* representa um dos valores abstratos atômicos (*impossible*, *ground-deref*, *ground*, *uninit*, *free*, *deref* e *any*)".

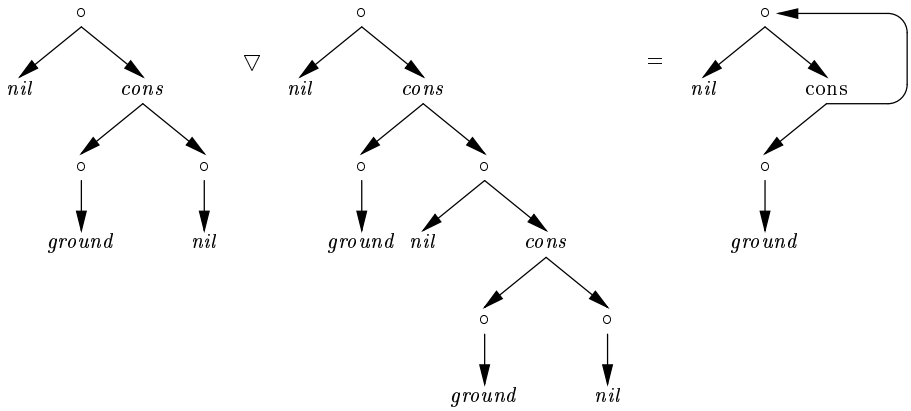

Figura 7.14: Exemplo de uso do operador $\triangledown$

1. Inicialmente o analisador assume que o efeito de execução é o predicado divergente, logo temos que:
$$append(ground \to \bot, ground \to \bot, uninit \to \bot)$$

2. Após a primeira iteração temos que:
$$append(ground \to ground, ground \to ground, uninit \to nil)$$

3. Após a segunda iteração temos que:
$$append(ground \to ground, ground \to ground, uninit \to n_{f_2})$$
onde n_{f_2} é o terceiro grafo da seqüencia descrita na Figura 7.12.

4. Após a terceira iteração temos que:
$$append(ground \to ground, ground \to ground, uninit \to n_{f_3})$$
onde $n_{f_3} = n_{f_2} \triangledown n'_f$ e n'_f é o quarto grafo da seqüencia descrita na Figura 7.12. n_{f_3} é o grafo resultante contido na Figura 7.14.

5. Na quarta iteração o algoritmo converge, e o resultado é o grado n_{f_3}.

O analisador de *modo* também reordena os átomos de uma cláusula. Esta reordenação tem a função de melhorar a eficiência da código gerado. Este processo de reordenação é baseado na heurística de que é vantajoso realizar primeiro as unificações dos elementos mais instanciados, visto que, a chances de descartar uma alternativa infrutífera são maiores. Por exemplo, considere os seguintes exemplos de unificação:

- *ground-deref* = *ground-deref*: neste caso, a unificação funciona como um teste que verifica se as duas estruturas de dados são iguais. Este teste deve ser realizado o mais cedo possível, se este falhar a alternativa corrente é descartada imediatamente. Como os termos *ground-deref* não precisam ser dereferenciados, este tipo de unificação possui uma prioridade maior que unificações do tipo *ground-deref* = *ground*, *ground* = *ground-deref* ou *ground* = *ground*.

- *ground-deref* = *uninit*: este tipo de unificação sempre é bem sucedida e não causa nenhum problema.

```
append V0 V1 V2 :-
   (V0 ?= nil,
    V2 := V1
   )
   ;
   (V0 ?= (V3 :: V4),
    V5 := V3,
    V7 := V4,
    V8 := V1,
    (append V7 V8 V9)
    V6 := V9,
    V2 != (V5 :: V6),
   )
```

Figura 7.15: *append*(*ground* → *ground*, *ground* → *ground*, *uninit* → *ground*)

- *uninit* = *uninit*: este tipo de unificação deve ser sempre evitado, já que transforma elementos *uninit* em *free*. Quando uma unificação deste tipo ocorre, uma variável não inicializa tem que ser inicializada, e além disso, um *alias* é definido (i.e. cadeias de ponteiros são formadas).

- $\top = \top$: definitivamente este é o pior tipo de unificação, sendo necessária a utilização do algoritmo genérico descrito na Figura 7.2.

Os casos descritos acima estão longe de abranger todas as possibilidades consideradas pelo algoritmo de reordenação de átomos. Contudo, eles apresentam o "espírito" do nosso reordenador. A Figura 7.15 contém o predicado *append* após a análise de *modo*.

A *Uniqueness Analysis* deve ser executa em conjunto com a análise de *modo*. Entretanto, como mencionado no capítulo anterior, esta análise não esta presente na implementação atual. A ausência deste analisador não compromete a eficiência do código gerado, uma vez que a sua função é apenas gerar mensagens de "alerta" para invocações indevidas de predicados contendo marcações `unq`.

7.4 *Case Analysis*

Este analisador é trivial e funciona de forma puramente sintática. Nenhum dos algoritmos de análise apresentados nos capítulos anteriores precisam ser utilizado. Em suma, este "analisador" procura nas alternativas de uma cláusula alguma condição que possa ser utilizada para decidir qual das alternativas deve ser executada em tempo de execução.

O caso mais simples consiste em cláusulas que possuam apenas duas alternativas, e uma delas possua o átomo `P` e a outra o átomo `not P`. A Figura 7.16 contém um exemplo deste tipo de cláusula. Obviamente, este tipo de cláusula pode ser substituído por um *if-then-else*. A Figura 7.17 contém a cláusula da Figura 7.16 após a transformação.

Este analisador também procura nas alternativas por unificações de acesso, ou seja, unificações que acessam campos de um funtor. Enfim, estas unificações tem sucesso, se e somente se, o funtor acessado possui o *tag* esperado. Por exemplo, a primeira alternativa do predicado *append* contém a unificação `V0 ?= nil` e a segunda alternativa a unificação `V0 ?= (V3 :: V4)`.

```
tst V0 V1 :-
    ( V0 > 0,
      V1 is V0 + 1
    )
    ;
    ( not (V0 > 0),
      V1 is 1 - V0
    ).
```

Figura 7.16: Caso simples para o analisador de *casos*

```
tst V0 V1 :-
     if V0 > 0
     then (V1 is V0 + 1)
     else (V1 is 1 - V0)
```

Figura 7.17: Caso simples após a transformação

Ao analisarmos o *tag* do funtor referenciado pela variável `V0`, podemos então decidir qual alternativa deve ser executada. Se o *tag* for `nil` a primeira alternativa deve ser executada, e se o *tag* for `::` (*cons*) a segunda alternativa deve ser executada. Na Figura 7.15 mostra o predicado *append* após a execução do analisador de *casos*. Observe que a unificação `V0 ?= nil` foi removida da primeira alternativa, porque a sua função era apenas testar se o *tag* era *nil*.

Em implementações futuras, pretendemos aprimorar o analisador de casos com o intuito de identificar outras condições que possam determinar precisamente qual alternativa deve ser executada. Particularmente, é interessante identificar átomos envolvendo condições mutuamente exclusivas tais como `V0 > 0` e `V0 <= 0`, e condições compostas envolvendo valor de *tags*, expressões aritméticas, etc. [5].

7.5 *Determinism Analysis*

O número de soluções retornadas por um *goal* é fundamental para a geração de código eficiente. Por exemplo, se um *goal* é determinístico, i.e. possui exatamente uma solução, então o compilador não precisa criar contexto para um eventual *backtracking*.

Em geral, o número de soluções de um *goal* associado a um predicado depende de como este é invocado. Por isso, é importante realizar a análise de determinismo, após a análise de modo, que categoriza os *modos* de uso de um predicado. Portanto, estamos na verdade analisando o determinismo dos *modos* de um predicado, e não do predicado propriamente dito.

A execução de um *goal* pode ter sucesso, falhar ou entrar em *loop* (divergir). O caso mais simples de *goal* que sempre diverge é *simple-loop*, que está associado ao predicado:

```
simple-loop :- simple-loop.
```

[5] O analisador atual identifica condições envolvendo mais de um *tag*.

```
append V0 V1 V2 :-
   case tag(V0)
   nil:
       (V2 := V1)
   cons:
       (V0 ?= (V3 :: V4),
        V5 := V3,
        V7 := V4,
        V8 := V1,
        (append V7 V8 V9)
        V6 := V9,
        V2 != (V5 :: V6),
       )
```

Figura 7.18: $append(ground \rightarrow ground, ground \rightarrow ground, uninit \rightarrow ground)$

Um *goal* pode, também, executar com sucesso um número x de vezes e divergir em seguida. Por exemplo, o *goal "tst X"* associado ao predicado *tst*, descrito a seguir, executa com sucesso duas vezes, e então entra em *loop*.

```
tst 1.
tst 2.
tst X :- simple-loop.
```

Portanto, a análise exata do número de soluções de um *goal*, deveria considerar os seguintes casos:

0 : o *modo* sempre falha.

1 : o *modo* executa com sucesso uma vez.

2 : o *modo* executa com sucesso duas vezes.

3 : o *modo* executa com sucesso três vezes.

...

$\perp$: o *modo* entra em *loop*.

$1^{\perp}$: o *modo* executa com sucesso uma vez, e então entra em *loop*.

$2^{\perp}$: o *modo* executa com sucesso duas vezes, e então entra em *loop*.

$3^{\perp}$: o *modo* executa com sucesso três vezes, e então entra em *loop*.

...

Observe que não analisamos *goals*, mas sim os *modos* de um predicado. Conseqüentemente, o número de soluções atribuído a um modo de predicado, representa uma aproximação do número de soluções de qualquer *goal* associado a este *modo* de predicado. Por isso, nosso analisador

deve, a princípio, associar a cada *modo* de predicado A um subconjunto A_{sols} do conjunto $\{0, 1, 2, 3, \ldots, \perp, 1^{\perp}, 2^{\perp}, 3^{\perp}, \ldots\}$.

Em geral, não estamos interessados no número exato de soluções de um *goal*. Desta forma, podemos fazer a seguinte simplificação:

- $0 \in A_{sols}$: "*A pode* não conter soluções ao se realizar uma busca exaustiva"
- $1 \in A_{sols}$: "*A pode* conter apenas uma solução ao se realizar uma busca exaustiva"
- $2 \in A_{sols}$: "*A pode* conter apenas duas ou mais soluções ao se realizar uma busca exaustiva"
- $\perp \in A_{sols}$: "*A pode* entrar em *loop* ao se realizar uma busca exaustiva"
- $1^{\perp} \in A_{sols}$: "*A pode* retornar uma solução e entrar em *loop* em seguida ao se realizar uma busca exaustiva"
- $2^{\perp} \in A_{sols}$: "*A pode* retornar duas ou mais soluções e entrar em *loop* em seguida ao se realizar uma busca exaustiva"

Note que em todos os casos utilizamos a palavra *pode*, já que o conjunto A_{sols} pode conter mais de um elemento. Por exemplo $A_{sols} = \{0, 1\}$ representa o caso onde durante uma busca exaustiva, *A pode* não conter soluções, *ou pode* conter apenas uma solução.

O caso de um *goal* contendo infinitas soluções pode ser representado tanto por 2 como por $2^{\perp}$, mas em nossa implementação utilizamos $2^{\perp}$ para representar o caso de infinitas soluções.

Todos os predicados *built-in* são pré-classificados em nosso analisador. Os predicados implementados em *C* devem ser classificados pelo usuário, conforme foi descrito no Capítulo 6. A seguir descrevemos a classificação de alguns predicados *built-in*.

fail : $\{0\}$

length(in, out) : $\{1\}$

length(in, in) : $\{0, 1\}$

length(out, out) : $\{2^{\perp}\}$

O predicado *length* pode ser descrito em *PAN* como:

```
type length : list A -> int -> o.
length nil 0.
length (X :: Xs) N :-
   length Xs N1,
   N is N1 + 1.
```

Em geral, o conjunto solução de uma unificação "$X = Y$" é $\{0, 1\}$. Entretanto se o analisador de modo marcou uma unificação como sendo uma atribuição (:=) ou uma alocação (!=), então o conjunto solução será $\{1\}$. Esta é mais uma justificativa para realizarmos a análise de determinismo após a análise de modo. Além disso, se o analisador de modo marcou uma unificação como sendo uma "deconstrução" (?=), e esta unificação está dentro de um comando "*case*", que garante que a "deconstrução" terá sempre sucesso, então o conjunto solução também será $\{1\}$. Dizemos que um comando "*case*" garante o sucesso de uma "deconstrução", quando este verifica o *tag* da variável que está sendo "deconstruída". Por exemplo, a "deconstrução" `(V0 ?= (V3 :: V4))` da Figura 7.18 terá sempre sucesso, já que esta se encontra dentro de um comando "*case*" que testa o *tag* de `V0`.

Nosso compilador provê as seguintes marcações para classificar predicados implementados em *C*.

det : $\{1\}$

semi-det : $\{0, 1\}$

multi : $\{2\}$

non-det : $\{0, 2\}$

any : $\{0, 1, 2\}$

non-det* : $\{0, 2, 2^{\perp}\}$

any* : $\{0, 1, 2, \perp, 1^{\perp}, 2^{\perp}, \}$

Obviamente, os marcadores providos não esgotam todas as possibilidades, mas são suficientes para uma classificação satisfatória da maioria dos predicados. Note que, a maioria dos marcadores ignora o caso de divergência. Somente os marcadores terminados em "*" consideram esta possibilidade. A marcação *any** pode ser sempre utilizada, visto que ela é uma aproximação segura para qualquer predicado. Entretanto, ao utilizarmos a marcação *any** o compilador pouco poderá fazer para otimizar o código, visto que esta marcação não diz "nada", i.e. ela representa o elemento $\top$ do reticulado de aproximações.

O "!" (*cut*) complica um pouco a nossa análise, visto que este modifica diretamente o número de soluções de um predicado. Para lidarmos com o *cut*, criamos duas versões para cada classificação de número de soluções. Por exemplo, a classificação 1_n representa uma solução onde nenhum *cut* foi executado e 1_c representa uma solução onde pelo menos um *cut* foi executado. Desta maneira, temos que o conjunto de possíveis classificações é $\{0_n, 1_n, 2_n, \perp_n, 1_n^{\perp}, 2_n^{\perp}, 0_c, 1_c, 2_c, \perp_c, 1_c^{\perp}, 2_c^{\perp}\}$. As marcações utilizadas para classificar predicados implementados em C devem ser interpretadas como subconjuntos de $\{0_n, 1_n, 2_n, \perp_n, 1_n^{\perp}, 2_n^{\perp}\}$, visto que este tipo de predicado, por motivos óbvios, nunca realiza um *cut*.

O subconjunto de classificações de um determinado *modo* de predicado é calculado a partir da sua estrutura. A seguir, descrevemos como é realizado o cálculo para os principais construtores da linguagem *PAN*.

7.5.1 Disjunção

A disjunção é simples de ser analisada. Abstratamente, o número de soluções de $s_1; s_2$ é igual ao número de soluções de s_1 "+" o número de soluções de s_2. Por exemplo, se s_1 executa com sucesso exatamente uma vez, e s_2 sempre falha, então $s_1; s_2$ executa com sucesso exatamente uma vez.

Para facilitar a exposição, não iremos, a princípio, considerar o *cut*. A Figura 7.19 contem a definição do operador "+" de número de soluções.

Como descrito anteriormente a classificação de um *modo* de predicado é formado por um subconjunto dos elementos $\{0, 1, 2, \perp, 1^{\perp}, 2^{\perp}\}$. Portanto, a função "+" deve ser extendida para tratar conjuntos. Assim, temos a seguinte definição.

$$N_{;}(S_1, S_2) = \{s_1 + s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\}$$

Então temos que o conjunto solução para uma disjunção "*A;B*" é:

$$(A; B)_{sols} = N_{;}(A_{sols}, B_{sols})$$

Esta solução assume que A e B não contêm *cuts*. Para considerarmos *cuts*, devemos adaptar a função "+" da seguinte maneira:

s_1 \ s_2	$+$	0	1	2	$\perp$	$1^\perp$	$2^\perp$
$+$	0	1	2	$\perp$	$1^\perp$	$2^\perp$
0	0	1	2	$\perp$	$1^\perp$	$2^\perp$
1	1	2	2	$1^\perp$	$2^\perp$	$2^\perp$
2	2	2	2	$2^\perp$	$2^\perp$	$2^\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$1^\perp$	$1^\perp$	$1^\perp$	$1^\perp$	$1^\perp$	$1^\perp$	$1^\perp$
$2^\perp$	$2^\perp$	$2^\perp$	$2^\perp$	$2^\perp$	$2^\perp$	$2^\perp$

Figura 7.19: Definição de $s_1 + s_2$ (disjunção)

$$\begin{aligned} a_c + b_c &= a_c \\ a_c + b_n &= a_c \\ a_n + b_c &= (a+b)_c \\ a_n + b_n &= (a+b)_n \end{aligned}$$

Nos dois primeiros casos, o número de soluções da segunda parte da disjunção é ignorado, visto que pelo menos um *cut* foi executado na primeira parte. Nos dois últimos casos, nenhum *cut* foi executado pela primeira parte da disjunção, logo o número de soluções é "somado" normalmente.

7.5.2 Conjunção

A conjunção é mais complexa de ser analisada. Diversas condições devem ser checadas para verificar se um determinado elemento pertence ou não ao conjunto solução. A seguir descrevemos todos os casos possíveis. Em alguns casos, adicionamos comentários para facilitar a compreensão. Usamos a notação "x_i" para representar os elementos "x_n" e "x_c".

- $0_n \in (A,B)_{sols}$ se
  - $0_n \in A_{sols}$, ou
  - (1_n ou $2_n \in A_{sols}$ "A não entra necessariamente em *loop*") e $0_n \in B_{sols}$
- $1_n \in (A,B)_{sols}$ se
  - $1_n \in A_{sols}$ e $1_n \in B_{sols}$, ou
  - $2_n \in A_{sols}$ e $0_n \in B_{sols}$ e $1_n \in B_{sols}$ ("A pode ser executado com sucesso 2 ou mais vezes, mas B falha em todos os casos menos um, i.e. se A executou com sucesso x vezes, mas B falhou $x-1$ e teve sucesso 1 vez.")
- $2_n \in (A,B)_{sols}$ se
  - $2_n \in A_{sols}$ e $1_n \in B_{sols}$, ou
  - (1_n ou $2_n \in A_{sols}$) e $2_n \in B_{sols}$
- $\perp_n \in (A,B)_{sols}$ se

- $\bot_n \in A_{sols}$, ou
- ($1_n^\bot$ ou $2_n^\bot \in A_{sols}$) e $0_n \in B_{sols}$ ("A, B entra em *loop*, porque B pode sempre falhar e conseqüentemente forçar que todas as soluções de A sejam tentadas até este entrar em *loop*"), ou
- (1_i ou 2_i ou $1_i^\bot$ ou $2_i^\bot \in A_{sols}$) e $\bot_n \in B_{sols}$ ("A executa com sucesso pelo menos uma vez, e B entra imediatamente em *loop*"). Também estamos considerando os caso (1_c ou 2_c ou $1_c^\bot$ ou $2_c^\bot \in A_{sols}$), já que o *cut* pode ser executado apenas quando ocorre um *backtracking*. Por exemplo, "*(true ; (!, fail)), simple-loop*" representa esta situação, onde $A_{sols} = \{1_c\}$ e $B_{sols} = \{\bot_n\}$, mas $(A, B)_{sols} = \{\bot_n\}$, i.e. durante a execução exaustiva, (A, B) entrou em *loop* sem executar nenhum *cut*. A seguir mostramos passo-a-passo como verificar este resultado:

$$\begin{array}{lcl} !_{sols} & = & \{1_c\} \\ fail_{sols} & = & \{0_n\} \\ (!, fail) & = & \{0_c\} \\ true_{sols} & = & \{1_n\} \\ (true; (!, fail))_{sols} & = & \{1_c\} \\ simple\text{-}loop_{sols} & = & \{\bot_n\} \\ ((true; (!, fail)), simple\text{-}loop)_{sols} & = & \{\bot_n\} \end{array}$$

- $1_n^\bot \in (A, B)_{sols}$ se
  - (1_i ou 2_i ou $1_i^\bot$ ou $2_i^\bot \in A_{sols}$) e $1_n^\bot \in B_{sols}$ ("Semelhante ao item anterior"), ou
  - $1_n^\bot \in A_{sols}$ e $1_n \in B_{sols}$, ou
  - $2_n^\bot \in A_{sols}$ e $0_n \in B_{sols}$ e $1_n \in B_{sols}$, ou
  - (2_i ou $2_i^\bot \in A_{sols}$) e $1_n \in B_{sols}$ e $\bot_n \in B_{sols}$. Para exemplificar esta situação assuma que X é uma variável livre antes da execução no trecho de código C a seguir.

```
(X = 1; X = 2; (X = 3, !, simple-loop)),
(X = 1 ; (X = 2, simple-loop)).
```

O analisador de *modo* irá assinalar as duas primeiras unificações como atribuições, porque X é uma variável livre antes de executar o trecho de código. A última unificação é assinalada como um teste, já que neste ponto X é uma variável "ligada", logo temos que:

```
(X := 1; X := 2; (X := 3, !, simple-loop)),
 (X == 1 ; (X == 2 ,  simple-loop)).
```

Então $C_{sols} = \{1_n^\bot\}$. A seguir mostramos como verificar passo-a-passo este resultado:

$$\begin{array}{lcl} (X := n)_{sols} & = & \{1_n\} \\ (X := 3, !, simple\text{-}loop) & = & \{\bot_c\} \\ (X == n)_{sols} & = & \{0_n, 1_n\} \\ (X == 1; (X == 2, simple\text{-}loop))_{sols} & = & \{1_n, \bot_n\} \end{array}$$

É importante lembrar que uma atribuição nunca falha, mas um teste pode ou não falhar.

- $2_n^\bot \in (A, B)_{sols}$ se

- (1_i ou 2_i ou $1_i^\perp$ ou $2_i^\perp \in A_{sols}$) e $2_n^\perp \in B_{sols}$, ou
- ($1_n^\perp$ ou $2_n^\perp \in A_{sols}$) e $2_n \in B_{sols}$, ou
- $2_n^\perp \in A_{sols}$ e $1_n \in B_{sols}$, ou
- $2_n^\perp \in A_{sols}$ e $0_n \in B_{sols}$ e (2_n ou $1_n \in B_{sols}$) (Condição redundante, compare-a com as condições definidas nos dois ítens anteriores. Entretanto, ela define uma situação completamente diferente), ou
- $2_i \in A_{sols}$ e ($1_n \in B_{sols}$) e ($1_n^\perp \in B_{sols}$), ou
  Exemplo:

  ```
  (X := 1; X := 2),
  (X == 1; (X == 2, (true  ; simple-loop))).
  ```

- $2_i \in A_{sols}$ e ($2_n \in B_{sols}$) e ($\perp_n$ ou $1_n^\perp \in B_{sols}$)
  Exemplo:

  ```
  (X := 1; (!, X := 2) ; X := 4),
  ((X == 1, (true; true)) ; ( X == 2; simple_loop)).
  ```

- $0_c \in (A, B)_{sols}$ se
  - $0_c \in A_{sols}$, ou
  - (1_n ou 2_n ou $1_i^\perp$ ou $2_i^\perp \in A_{sols}$) e $0_c \in B_{sols}$ ("O *cut* em B evita que mais de uma alternativa seja tentada em A, evitando que A entre em *loop*"), ou
  - (1_c ou $2_c \in A_{sols}$) e $0_i \in B_{sols}$
- $1_c \in (A, B)_{sols}$ se
  - $1_c \in A_{sols}$ e $1_n \in B_{sols}$, ou
  - (1_i ou 2_i ou $1_i^\perp$ ou $2_i^\perp \in A_{sols}$) e $1_c \in B_{sols}$, ou
  - (2_i ou $2_i^\perp \in A_{sols}$) e $0_n \in B_{sols}$ e $1_c \in B_{sols}$ (Note que esta condição é desnecessária, já que sempre que esta for válida a condição definida no item anterior também será válida. Por outro lado, esta condição ilustra o caso onde A executa com sucesso x vezes e B falha em seguida sem executar o *cut*; na tentativa $x + 1$, B executa com sucesso e realiza o *cut*, impedindo que A seja executado novamente.), ou
  - ($2_c \in A_{sols}$) e $0_n \in B_{sols}$ e $1_n \in B_{sols}$.
    Exemplo:

    ```
    (X := 1; (!, X := 2) ; X := 4),
    (X == 2).
    ```

- $2_c \in (A, B)_{sols}$ se
  - $2_c \in A_{sols}$ e $1_n \in B_{sols}$, ou
  - (1_c ou $2_c \in A_{sols}$) e $2_n \in B_{sols}$, ou
  - (1_i ou 2_i ou $1_i^\perp$ ou $2_i^\perp \in A_{sols}$) e $2_c \in B_{sols}$
- $\perp_c \in (A, B)_{sols}$ se
  - $\perp_c \in A_{sols}$, ou

- $(1_i$ ou 2_i ou $1_i^\perp$ ou $2_i^\perp \in A_{sols})$ e $\perp_c \in B_{sols}$, ou
- $(1_c$ ou 2_c ou $1_c^\perp$ ou $2_c^\perp \in A_{sols})$ e $\perp_n \in B_{sols}$, ou
- $(1_c^\perp$ ou $2_c^\perp \in A_{sols})$ e $0_n \in B_{sols}$

- $1_c^\perp \in (A, B)_{sols}$ se
  - $(1_i$ ou 2_i ou $1_i^\perp$ ou $2_i^\perp \in A_{sols})$ e $1_c^\perp \in B_{sols}$, ou
  - $(1_c$ ou 2_c ou $1_c^\perp$ ou $2_c^\perp \in A_{sols})$ e $1_n^\perp \in B_{sols}$, ou
  - $1_c^\perp \in A_{sols}$ e $1_n \in B_{sols}$, ou
  - $2_c^\perp \in A_{sols}$ e $0_n \in B_{sols}$ e $1_n \in B_{sols}$, ou
  - $(2_i$ ou $2_i^\perp \in A_{sols})$ e $1_n \in B_{sols}$ e $\perp_c \in B_{sols}$, ou
  - $(2_c$ ou $2_c^\perp \in A_{sols})$ e $1_n \in B_{sols}$ e $\perp_n \in B_{sols}$

- $2_c^\perp \in (A, B)_{sols}$ se
  - $(1_i$ ou 2_i ou $1_i^\perp$ ou $2_i^\perp \in A_{sols})$ e $2_c^\perp \in B_{sols}$, ou
  - $(1_c$ ou 2_c ou $1_c^\perp$ ou $2_c^\perp \in A_{sols})$ e $2_n^\perp \in B_{sols}$, ou
  - $(1_c^\perp$ ou $2_c^\perp \in A_{sols})$ e $2_n \in B_{sols}$, ou
  - $2_c^\perp \in A_{sols}$ e $1_n \in B_{sols}$, ou
  - $2_c \in A_{sols}$ e $1_n \in B_{sols}$ e $1_i^\perp \in B_{sols}$, ou
  - $2_i \in A_{sols}$ e $1_n \in B_{sols}$ e $1_c^\perp \in B_{sols}$, ou
  - $(2_c^\perp \in A_{sols})$ e $0_n \in B_{sols}$ e $(1_n$ ou $2_n \in B_{sols})$ (Condição redundante, compare-a com as condições definidas nos ítens 3 e 4. Entretanto, define uma situação completamente diferente), ou
  - $(2_c$ ou $2_c^\perp \in A_{sols})$ e $(1_n$ ou $2_n \in B_{sols})$ e $\perp_n \in B_{sols}$, ou
  - $(2_i$ ou $2_i^\perp \in A_{sols})$ e $(1_n$ ou $2_n \in B_{sols})$ e $\perp_c \in B_{sols}$

7.5.3 Negação

A negação é mais simples de ser analisada. É importante lembrar, que os *cuts* existentes dentro de uma negação não tem efeito "fora" do escopo da negação. Uma negação pode executar com sucesso exatamente uma vez, falhar, ou entrar em *loop*. Portanto, o conjunto de soluções associada a uma negação é formado apenas pelos elementos $\{0_n, 1_n, \perp_n\}$. A seguir descrevemos as condições para que cada um destes elementos esteja no conjunto de soluções associado a uma negação:

- $1_n \in (not\ A)_{sols}$ se $0_i \in A_{sols}$
- $0_n \in (not\ A)_{sols}$ se $(1_i$ ou 2_i ou $1_i^\perp$ ou $2_i^\perp \in A_{sols})$
- $\perp_n \in (not\ A)_{sols}$ se $\perp_i \in A_{sols}$

7.5.4 If-Then-Else

O conjunto solução associado ao construtor *if-then-else* da linguagem *PAN* é definido como:

$$
\begin{aligned}
&(if\ Cond\ then\ ThenBody\ else\ ElseBody)_{sols} = \\
&\quad (se\ \perp_i \in (Cond)_{sols}\ então\ \{\perp_i\}\ senão\ \emptyset) \cup \\
&\quad (se\ (1_i\ ou\ 1_i^\perp\ ou\ 2_i\ ou\ 2_i^\perp \in (Cond)_{sols})\ então\ (ThenBody)_{sols}\ senão\ \emptyset) \cup \\
&\quad (se\ 0_i \in (Cond)_{sols}\ então\ (ElseBody)_{sols}\ senão\ \emptyset)
\end{aligned}
$$

7.5.5 Case

O construtor *case* não está presente na linguagem *PAN*, mas ele é adicionado pela *case analysis*. Portanto, este construtor deve ser considerado pelo nosso analisador de determinismo. Felizmente, este caso também é o mais simples de todos. O conjunto solução associado ao construtor *case* consiste na união dos conjuntos solução associados a cada alternativa.

7.5.6 Recursão

Os predicados recursivos e mutuamente recursivos são analisados utilizando o algoritmo de análise interprocedural descrito na Seção 5.6.3.1. Neste caso, utilizamos o contexto vazio, simplificando, desta forma, o algoritmo de análise interprocedural. O efeito de execução descrito no algoritmo é definido neste caso como o subconjunto solução de classificação de determinismo.

Como foi visto na Seção 5.6.3.1, o algoritmo de análise interprocedural pode ser visto como um cálculo interativo do menor ponto fixo de um conjunto de equações recursivas. Conseqüentemente, para aplicarmos o algoritmo temos que definir uma noção de ordem entre os elementos abstratos (propriedades), i.e. os subconjuntos do conjunto $\{0_n, 1_n, 2_n, \bot_n, 1_n^{\bot}, 2_n^{\bot}\}$. As classificações relativas ao *cut* não devem ser consideradas na classificação de predicados (recursivos ou não), já que o escopo de um *cut* é restrito ao predicado onde o mesmo está contido.

Para definirmos uma noção de ordem, devemos observar que nem todos os subconjuntos de $\{0_n, 1_n, 2_n, \bot_n, 1_n^{\bot}, 2_n^{\bot}\}$ podem ser considerados. Por exemplo, o subconjunto $\{\}$ não possui significado algum como a classificação de um predicado. O elemento mínimo deve ser o subconjunto $\{\bot\}$, que correponde ao caso de um predicado que sempre diverge. A noção de ordem pode então ser definida pelo operados $\subseteq$ entre conjuntos. Mas surge, um problema, visto que como $\{\bot\}$ é o elemento mínimo, então a possibilidade de divergência deve estar presente em qualquer classificação associada a um predicado recursivo. Tal fato não é de todo estranho, visto que com as aproximações utilizadas não é possível determinar se um predicado é divergente ou não. Por exemplo, considere os dois predicados a seguir:

```
append nil X X.
append (X :: Xs) Ys (X :: Zs) :-
    append Xs Ys Zs.

bug-append nil X X.
bug-append (X :: Xs) Ys (X :: Zs) :-
    bug_append (X :: X :: Xs) Ys (X :: X :: Xs)
```

Como o nosso analisador trabalha com *modos* de predicado, assuma que os dois primeiros parâmetros estão completamente instanciados e o terceiro completamente livre. Logo, o predicado *append* sempre converge, mas o predicado *bug-append* somente converge quando o primeiro argumento for *nil*. Apesar destes predicados se comportarem de forma completamente diferente, estes são "vistos" da mesma forma pelo analisador de determinismo. Em ambos os casos o analisador irá obter o resultado $\{1_n, \bot_n\}$.

Para exemplificarmos o funcionamento do analisador, mostraremos a seguir a computação da solução para o predicado *append*. Utilizaremos como exemplo o *modo* descrito na Figura 7.18. A Figura 7.20 contem este mesmo modo, onde os *goals* atômicos foram substituídos por seus respectivos conjuntos solução. Para facilitar o entendimento, os *goals* que originaram cada subconjunto foram também descritos. Lembre que inicialmente consideramos que a solução de um predicado recursivo é $\{\bot_n\}$. A Figura 7.21 contem os resultados intermediários após a aplicação do operador associado a conjunção "," definido anteriormente. Usando então a definição do operador associado ao *case* podemos concluir que a solução do predicado recursivo *append*

```
append V0 V1 V2 :-
        case tag(V0)
        nil:
            {1_n} // (V2 := V1)
        cons:
            {1_n}, // V0 ?= (V3 :: V4)
            {1_n}, // V5 := V3
            {1_n}, // V7 := V4
            {1_n}, // V8 := V1
            {⊥_n}, // (append V7 V8 V9)
            {1_n}, // V6 := V9,
            {1_n} // V2 != (V5 :: V6)
```

Figura 7.20: Passo 1: cálculo do determinismo

```
append V0 V1 V2 :-
        case tag(V0)
        nil:
            {1_n}
        cons:
            {⊥_n}
```

Figura 7.21: Passo 2: cálculo do determinismo

é $\{1_n, \bot_n\}$. Como o solução anterior é diferente da solução obtida nesta iteração, temos que calcular a solução novamente, mas considerando que a solução do predicado *append* é $\{1_n, \bot_n\}$. Felizmente, o resultado desta nova iteração será novamente $\{1_n, \bot_n\}$.

Os resultados obtidos para predicados recursivos podem ser melhorados mediante ao uso de um analisador de terminação. O analisador de terminação verifica se um predicado converge ou não. Abstratamente, o analisador verifica apenas se os parâmetros de uma chamada recursiva estão diminuindo de "tamanho". Condição esta suficiente para garantir a terminação de um predicado. Obviamente, este analisador produz apenas resultados aproximados, i.e. um predicado pode ser erroneamente classificado como divergente.

Quando o analisador de terminação classifica um predicado como convergente, podemos então remover o elemento $\bot_n$ do conjunto solução produzido pelo analisador de determinismo. Podemos também, substituir os elementos $1_n^{\bot}$ e $2_n^{\bot}$ por 1_n e 2_n respectivamente.

7.6 Predicados de Alta Ordem X Analisadores

Predicados de alta ordem complicam a implementação dos analisadores de programas. O principal motivo é que em alguns casos pode não ser possível determinar qual predicado está sendo realmente invocado. Considere o exemplo a seguir.

```
type map : (A -> B -> o) -> list A -> list B -> o.
map F nil nil.
map F (X :: Xs) (Y :: Ys) :- (F X Y), map F Xs Ys.
```

```
type id : A -> A -> o.
id X X.

type inc : int -> int -> o.
inc X Y :- Y is X + 1.

type confuso : in int -> in list int -> out list int -> o.
confuso N Xs Ys :-
(if N > 0 then F = id else F = inc),
map F Xs Ys.
```

Neste caso, se o valor N recebido como parâmetro de entrada for um número positivo, então o predicado *map* aplicará o predicado *id* sobre cada elemento da lista Xs, caso contrário aplicará o predicado *inc*.

Para lidarmos com este problema, adotamos uma solução simples, porém eficaz em um grande número de casos. Antes de realizar qualquer análise, o nosso compilador cria sempre que possível instâncias particulares de predicados de alta ordem. Por exemplo, considere a seguinte definição alternativa do predicado *confuso*.

```
type confuso : in int -> in list int -> out list int -> o.
confuso N Xs Ys :-
(if N > 0 then
map id Xs Ys
 else
map inc Xs Ys)
```

Neste caso, nosso compilador cria duas versões do predicado *map*: *map-id* e *map-inc*. Estas duas novas versões são de primeira ordem e possuem a seguinte definição:

```
type map-id : list A -> list B -> o.
map-id nil nil.
map-id (X :: Xs) (Y :: Ys) :- (id X Y), map-id Xs Ys.

type map-inc : list A -> list B -> o.
map-inc nil nil.
map-inc (X :: Xs) (Y :: Ys) :- (inc X Y), map-inc Xs Ys.
```

Esta transformação pode ser vista como uma avaliação parcial [51] extremamente simples. As "chamadas" a predicados de alta-ordem que não puderem ser transformadas permanecerão inalteradas, e os analisadores assumiram a pior situação possível quando estiverem analisando este tipo de chamada. Por exemplo, o analisador de determinismo irá assumir que uma chamada a um predicado de alta ordem terá classificação *any**. Já o analisador de modos assumirá que o modo de saída de um parâmetro será *any*.

A vantagem desta abordagem é que a transformação ("avaliação parcial") está completamente desacoplada dos analisadores. Portanto, no futuro podemos implementar transformações mais sofisticadas sem ter que modificar os analisadores.

7.7 Modelo de Execução

A maioria das implementações de linguagens lógicas utilizam um mesmo algoritmo para executar todos os predicados, introduzindo focos de ineficiência desnecessários no código gerado.

Não é necessário, por exemplo, criar contexto para um eventual *backtracking* de um predicado que nunca falha. Desta forma, utilizamos uma abordagem semelhante a utilizada no compilador da linguagem *Mercury* [91]. Este modelo explora a informação obtida pelo analisador de *determinismo*. O nosso modelo de execução possui três algoritmos de execução: um para predicados determinísticos, um para predicados não determinísticos e um para predicados semi-determinísticos [6].

7.7.1 Usando *C* como *Assembly*

Em nossa implementação não estamos interessados em gerar código *assembly* diretamente, já que isto nos prenderia a uma plataforma particular. Decidimos, então, utilizar a linguagem *C* como um *assembly* portável. Além disso, decidimos utilizar recursos adicionais [7] existentes no compilador *GNU C*. As seguintes extensões providas pelo *GNU C* são utilizadas em nosso compilador:

- Referência a *labels* do programa. O *GNU C* permite que uma variável referencie um *label* do programa. Por exemplo, o comando `currip = &&label1;` faz a variável `currip` referenciar o *label* `label1`, e `goto *currip;` representa um salto para o *label* referenciado por `currip`. Este recurso é fundamental para uma implementação eficiente do *backtracking*.

- Associação de registradores da máquina às variáveis do programa. Por exemplo, a declaração `register int r1 __asm__("ESI");` Este recurso permite que associemos registradores de nossa máquina virtual com registradores da máquina. Obviamente, esta associação é dependente de máquina. De qualquer maneira, isto não é um grande problema, visto que os comandos que definem a associação fazem parte de um preâmbulo que pode ou não ser incluído no código gerado por nosso compilador. Para máquinas que estas associações estejam disponíveis, o código gerado será mais eficiente.

No futuro, pretendemos gerar código para a linguagem $C--$ [53], que é uma simplificação da linguagem *C* especialmente desenvolvida para funcionar como um *assembly* portável. A linguagem $C--$ não é um subconjunto de *C*, da mesma foram que diversos recursos de *C* foram removidos, outros foram adicionados, como por exemplo um coletor de lixo. Infelizmente a linguagem $C--$ ainda não está completamente definida.

7.7.2 Representação dos dados

Como conhecemos todos os tipos em tempo de compilação, podemos especializar a representação dos termos para cada tipo, diminuindo, desta maneira, o consumo de memória e aumentando a eficiência do código.

De forma geral, um termo é composto por um *tag* e referências para os seus componentes. Por exemplos o termo `h :: t` possui um *tag* identificando o funtor `::` (*cons*) e as referências para os termos `h` e `t`.

A partir desta representação genérica, podemos instanciar os casos particulares. Por exemplo, *termos* de tipos que possuam apenas um funtor não precisam de *tags*, porque o tipo os identifica univocamente. *Termos* de tipos que possuam no máximo quatro funtores podem ter seu *tag* codificado nos dois bits menos significativos do ponteiro que o referencia. Isto é possível, já que a maioria dos computadores utiliza endereços de memória que são múltiplos de quatro.

[6] Um predicado é semi-determinístico, se este pode executar com sucesso no máximo uma vez, i.e. este pode falhar ou executar com sucesso uma vez.

[7] recursos que não fazem parte da definição da linguagem *C*.

7.7.3 Algoritmos de execução

Predicados classificados como $\{1_n\}$ ou $\{1_n, \perp_n\}$ comportam-se da mesma forma que procedimentos implementados em linguagens imperativas. Nossa implementação utiliza a abordagem convencional, com pilha de execução, registro de ativação e heap.

Entretanto, a abordagem convencional não pode ser utilizada para predicados que podem executar com sucesso mais de uma vez. O problema é o mecanismo de *backtracking*. Quando um *backtracking* ocorre, os valores das variáveis de um predicado podem não estar mais na pilha de execução. Por exemplo, considere o programa a seguir:

```
type p : int -> int -> o.
p(X,Y) :- (Y is X + 1) ; (Y is X + 2).

entry run : out int -> o.
run X :- p(2,Y), Y = 3, X = Y.
```

Ao executarmos o predicado *run*, este imediatamente invoca o predicado *p*. O predicado *p* executa `Y is X + 1` e retorna. O controle então retorna ao predicado *run*, que falha ao executar `Y = 3`. Esta falha gera um *backtracking*, forçando o controle a retornar ao predicado *p*.

Para resolver este problema, predicados não determinísticos utilizam uma pilha adicional denominada de *nondet stack*. Na verdade esta estrutura de dados não é realmente uma pilha, visto que a mesma pode crescer em qualquer direção. Para lidar com esta "pilha", utilizamos duas variáveis:

curfr: aponta para o registro de ativação (*frame*) corrente na *nondet stack*.

maxfr: aponta para o registro de ativação (*frame*) no topo da *nondet stack*.

Cada registro de ativação possui os seguintes dados:

succip: endereço de retorno quando o predicado executa com sucesso.

redoip: endereço de retorno quando a alternativa corrente falha. Isto é, aponta para a próxima alternativa, quando uma existe.

succfr: o valor de *curfr* a ser restaurado quando o predicado executa com sucesso. Aponta para um registro de ativação da *nondet stack*, quando o invocador é um predicado não determinístico.

prevfr: o valor de *curfr* a ser restaurado quando o predicado falha.

trail-ptr: quando uma variável recebe a marcação *any* pelo analisador de *modos*, não sabemos a sua instanciação em tempo de execução em um determinado ponto do programa. Logo, sempre que ocorre uma atribuição a uma variável livre marcada como *any*, é adicionada uma marcação dentro da pilha de *trail*. A pilha de *trail* contem o endereço das variáveis que devem ser transformadas em variáveis livres novamente, quando um *backtracking* ocorrer. O campo *trail-ptr* indica a posição da pilha onde as marcações associadas ao predicado corrente estão armazenadas.

framevars: zero ou mais variáveis locais.

Para exemplificar o funcionamento da *nondet-stack*, considere o programa a seguir:

Invocando *main X*
State 0
curfr → frame 0
maxfr → frame 0
nondet stack
frame 0
succip → nil
redoip → do-fail
prevfr → nil
succfr → nil

Invocando *tst X*
State 1
curfr → frame 1
maxfr → frame 1
nondet stack
frame 0
succip → nil
redoip → do-fail
prevfr → nil
succfr → nil
frame 1
succip → ②′
redoip → ③
prevfr → frame 0
succfr → frame 0

Retornando para *main X* em ②′
State 2
curfr → frame 0
maxfr → frame 1
Unificação X=5 falhou, já que X → 3
nondet stack
frame 0
succip → nil
redoip → do-fail
prevfr → nil
succfr → nil
frame 1
succip → ②′
redoip → ③
prevfr → frame 0
succfr → frame 0

Backtracking para *tst X* em ③
State 3
curfr → frame 1
maxfr → frame 1
nondet stack
frame 0
succip → nil
redoip → do-fail
prevfr → nil
succfr → nil
frame 1
succip → ②′
redoip → do-fail
prevfr → frame 0
succfr → frame 0

Retornando para *main X* em ②′
State 4
curfr → frame 0
maxfr → frame 1
Execução termina com sucesso
após unificação X=5
nondet stack
frame 0
succip → nil
redoip → do-fail
prevfr → nil
succfr → nil
frame 1
succip → ②′
redoip → do-fail
prevfr → frame 0
succfr → frame 0

Figura 7.22: Trace de execução de *main X*

tst X :-
① X = 3 ②
;
③ X = 5 ④.

main :-
①′ tst X,
②′ X = 5
③′.

Os símbolos ⓘ foram utilizados para marcar pontos de controle do programa. A Figura 7.22 mostra passo a passo os principais estados da computação do goal *main X*. A marcação *do-fail* representa o caso onde um predicado não possui mais alternativas a serem executadas. Observe que não estamos representado os campos *framevars* e *trail* para não complicar ainda mais o diagrama.

A seguir descrevemos um programa um pouco mais complexo. Note que, não consideramos as modificações realizadas pelos analisadores, afim de não prejudicar a compreensão. Apenas a conversão para *SHF* foi realizada. As Figuras 7.23, 7.24, 7.25 e 7.26 contêm o *trace* de execução

do goal *members (1::2::2::1::nil) Ys*. A Figura 7.27 contém o estado da *nondet stack* após a execução do goal. Se uma nova solução para o *goal* for requisitada, então o *frame* corrente passa a ser o 10. Como *(frame 10)→redoip* é igual a *do-fail*, devemos ir para *(frame 10)→prevfr*, cujo o valor e 9. Como *(frame 9)→redoip* é igual a ②, a computação recomeça no ponto de controle ②.

```
member V0 V1 :-
        ( ① V1 = V0 :: V2
        ) ;
        ( ② V1 = V3 :: V4,
             member V0 V4
          ③
        ).

members V0 V1 :-
        (
          ①' V0 = V2 :: V3,
             member V2 V1,
          ②' members V3 V1
          ③'
        ) ;
        ( ④' V0 = nil
          ⑤'
        ).
```

Tomando como base o exemplo anterior, fica claro que a implementação do *cut* consiste em fazer *maxfr := curfr*, e modificar *redoip* do *frame* corrente igual a *do-fail*.

Predicados semi-determísticos, i.e. classificados como $\{0_n, 1_n\}$ ou $\{0_n, 1_n, \perp_n\}$, podem executar com sucesso no máximo uma vez. Como estes nunca sofreram *backtracking*, então as suas variáveis podem ser armazenadas na pilha convencional. Nossa implementação converte predicados semi-determinísticos em predicados determinísticos que retornam um *flag* indicando se a execução teve sucesso ou não. O predicado que tenha invocado um predicado semi-determinístico deve sempre verificar o valor deste *flag* para determinar as medidas necessárias.

7.8 Conclusão

A implementação da linguagem *PAN* sofreu fortes influências dos compiladores *Aquarius Prolog* [82, 83] e *Mercury* [91]. Estes dois compiladores possuem modelos de execução mais sofisticados do que o da máquina virtual "clássica" (*WAM* [4]) utilizada na compilação de programas lógicos.

Em relação ao compilador *Aquarius Prolog*, podemos citar as seguintes diferenças:

- Como a linguagem é *Prolog*, o compilador possui um analisador de tipos.
- Representações específicas para termos de um mesmo tipo não são utilizadas.
- Apesar do analisador de modos ser interprocedural como o da linguagem *PAN*, este considera apenas valores abstratos atômicos. O compilador não consegue inferir a instanciação de variáveis parcialmente instanciadas.

Invocando
members (1::2::2::1::nil) Ys
State 0
curfr → frame 0
maxfr → frame 0

nondet stack
frame 0
succip → nil
redoip → ④
prevfr → nil
succfr → nil

Invocando *member 1 Ys*
State 1
curfr → frame 1
maxfr → frame 1
Unificação *(Ys = 1 :: P1)*
com sucesso.
P1 representa uma
nova variável livre.

nondet stack
frame 0
succip → nil
redoip → ④
prevfr → nil
succfr → nil

frame 1
succip → ②
redoip → ②
prevfr → frame 0
succfr → frame 0

Retornando para
members (1::2::2::1::nil) Ys
State 2
curfr → frame 0
maxfr → frame 1
Lembre que Ys = 1 :: P1.

nondet stack
frame 0
succip → nil
redoip → ④
prevfr → nil
succfr → nil

frame 1
succip → ②
redoip → ②
prevfr → frame 0
succfr → frame 0

Invocando
members (2::2::1::nil) (1 :: P1)
State 3
curfr → frame 2
maxfr → frame 2

nondet stack
frame 0
succip → nil
redoip → ④
prevfr → nil
succfr → nil

frame 1
succip → ②
redoip → ②
prevfr → frame 0
succfr → frame 0

frame 2
succip → ③
redoip → ④
prevfr → frame 1
succfr → frame 0

Invocando *member 2 (1 :: P1)*
State 4
curfr → frame 3
maxfr → frame 3
Unificação (1 :: P1) = (2 :: P2))
falha.

nondet stack
frame 0
succip → nil
redoip → ④
prevfr → nil
succfr → nil

...
frame 3
succip → ②
redoip → ②
prevfr → frame 2
succfr → frame 2

Backtracking member 2 (1 :: P1)
State 5
curfr → frame 3
maxfr → frame 3
Unificação (1 :: P1) = (P3 :: P4))
com sucesso.
P3 e P4 representam
novas variáveis.

nondet stack
frame 0
succip → nil
redoip → ④
prevfr → nil
succfr → nil

...
frame 3
succip → ②
redoip → do-fail
prevfr → frame 2
succfr → frame 2

Figura 7.23: Trace de execução de *members (1::2::2::1) Ys* 1/4

Invocando *member 2 P1*
State 6
curfr → frame 4
maxfr → frame 4
Unificação (P1 = (2 :: P5))
com sucesso.
P5 representa uma
nova variável.

nondet stack
...
frame 3
succip → ②'
redoip → do-fail
prevfr → frame 2
succfr → frame 2
frame 4
succip → ③
redoip → ②
prevfr → frame 3
succfr → frame 3

Retornando a *member 2 (1 :: P1)*
em ③
State 7
curfr → frame 3
maxfr → frame 4
Lembre que (P1 = 2 :: P5).

nondet stack
...
frame 3
succip → ②'
redoip → do-fail
prevfr → frame 2
succfr → frame 2
frame 4
succip → ③
redoip → ②
prevfr → frame 3
succfr → frame 3

Retornando a *member*
(2::2::1::nil) (1 :: P1) em ②'
State 8
curfr → frame 2
maxfr → frame 4
Lembre que (P1 = 2 :: P5)

nondet stack
...
frame 2
succip → ③'
redoip → ④'
prevfr → frame 1
succfr → frame 0
...
frame 4
succip → ③
redoip → ②
prevfr → frame 3
succfr → frame 3

Invocando *members*
(2::1::nil) (1 :: 2 :: P5)
State 9
curfr → frame 5
maxfr → frame 5

nondet stack
...
frame 5
succip → ③'
redoip → ④'
prevfr → frame 4
succfr → frame 2

Invocando *member* 2
(1 :: 2 :: P5)
State 10
curfr → frame 6
maxfr → frame 6
Unificação falha

nondet stack
...
frame 6
succip → ②'
redoip → ②
prevfr → frame 5
succfr → frame 5

Backtracking member 2
(1 :: 2 :: P5) em ②
State 11
curfr → frame 6
maxfr → frame 6
Unificação (2::1::P1) = (P6::P7)
com sucesso.

nondet stack
...
frame 6
succip → ②'
redoip → do-fail
prevfr → frame 5
succfr → frame 5

Figura 7.24: Trace de execução de *members (1::2::2::1) Ys* 2/4

Invocando *member 1 (1 :: P1)*
State 12
curfr → frame 7
maxfr → frame 7
Unificação (1::P1) = (1::P8)
com sucesso.

nondet stack
...
frame 6
succip → ②'
redoip → do-fail
prevfr → frame 5
succfr → frame 5
frame 7
succip → ③
redoip → ②
prevfr → frame 6
succfr → frame 6

Retornando a *member 2 (1::2::P1)*
em ③
State 13
curfr → frame 6
maxfr → frame 7

nondet stack
...
frame 6
succip → ②'
redoip → do-fail
prevfr → frame 5
succfr → frame 5
frame 7
succip → ③
redoip → ②
prevfr → frame 6
succfr → frame 6

Retornando a *member*
(2::2::1::nil) (1::2::P1) em ②'
State 14
curfr → frame 5
maxfr → frame 7

nondet stack
...
frame 5
succip → ③'
redoip → ④'
prevfr → frame 4
succfr → frame 2
...
frame 7
succip → ③
redoip → ②
prevfr → frame 6
succfr → frame 6

Invocando *members*
(1::nil) (1 :: 2 :: P1)
State 15
curfr → frame 8
maxfr → frame 8

nondet stack
...
frame 8
succip → ③'
redoip → ④'
prevfr → frame 7
succfr → frame 5

Invocando *member* 1
(1 :: 2 :: P1)
State 16
curfr → frame 9
maxfr → frame 9
Unificação (1::2::P1)=(1::P10)
com sucesso.

nondet stack
...
frame 9
succip → ②'
redoip → ②
prevfr → frame 8
succfr → frame 8

Retornando a *members* (1 :: nil)
(1 :: 2 :: P1) em ②
State 17
curfr → frame 8
maxfr → frame 9
Unificação (2::1::P1) = (P6::P7)
com sucesso.

nondet stack
...
frame 8
succip → ③'
redoip → ④'
prevfr → frame 7
succfr → frame 5
frame 9
succip → ②'
redoip → ②
prevfr → frame 8
succfr → frame 8

Figura 7.25: Trace de execução de *members (1::2::2::1) Ys* 3/4

Invocando *members nil (1::2::P1)*
State 18
curfr → frame 10
maxfr → frame 10
Unificação nil = (P11::P12)
falha.

nondet stack
...
frame 10
succip → ③'
redoip → ④'
prevfr → frame 9
succfr → frame 8

Backtracking members nil (1::2::P1) em ④'
State 19
curfr → frame 10
maxfr → frame 10
Unificação (nil=nil)
teve sucesso.

nondet stack
...
frame 10
succip → ③'
redoip → do-fail
prevfr → frame 9
succfr → frame 8

Retornando a *members (1::nil) (1::2::P1)*
State 20
curfr → frame 8
maxfr → frame 10

nondet stack
...

Retornando a *members (2::1::nil) (1::2::P1)*
State 21
curfr → frame 5
maxfr → frame 10

nondet stack
...

Retornando a *members (2::2::1::nil) (1::2::P1)*
State 22
curfr → frame 2
maxfr → frame 10

nondet stack
...

Retornando a *members (1::2::2::1::nil) (1::2::P1)*
State 23
curfr → frame 0
maxfr → frame 10
Fim da execução.

nondet stack
...

Figura 7.26: Trace de execução de *members (1::2::2::1) Ys* 4/4

- O compilador não gera diferentes versões de um mesmo predicado, i.e., uma para cada *modo* de uso.

Em relação ao compilador da linguagem *Mercury*, podemos citar as seguintes diferenças:

- Em *Mercury*, o usuário tem que fornecer declarações de *modo* além das declarações de tipo. O analisador de *modos* de *Mercury* está mais próximo de um *mode checker* do que de um analisador propriamente dito. Além disso, não é realizada nenhuma análise interprocedural.

- *Mercury* é menos expressiva que *Prolog*. Em *Mercury* não é permitido unificações entre duas variáveis livres. O analisador de *modo* tenta evitar ao máximo esta situação, e se a situação não puder ser evitada, um erro de compilação é gerado. Esta limitação de *Mercury* evita que vários programas *Prolog* sejam "traduzidos" para *Mercury*.

- O analisador de determinismo de *Mercury* está mais próximo de um *determinism checker* do que de um analisador propriamente dito. O nosso analisador também possui um reticulado de valores abstratos muito mais preciso.

- *Mercury* não possui predicados de alta ordem, e sim procedimentos de alta ordem. Onde um procedimento é definido como predicado mais declaração de modo. Esta limitação está relacionada a limitação do item anterior.

- *Mercury* não possui suporte a notação *Mixfix*.

Podemos dizer que o modelo de execução de *PAN* possui o esqueleto básico utilizado no compilador *Mercury*. Porém, foram adicionadas extensões para garantir que *PAN* tenha um poder expressivo mais próximo de *Prolog*. Tivemos, também, a intenção de poupar o usuário de

frame 0
succip → nil
redoip → ④′
prevfr → nil
succfr → nil

frame 1
succip → ②′
redoip → ②
prevfr → frame 0
succfr → frame 0

frame 2
succip → ③′
redoip → ④′
prevfr → frame 1
succfr → frame 0

frame 3
succip → ②′
redoip → do-fail
prevfr → frame 2
succfr → frame 2

frame 4
succip → ③
redoip → ②
prevfr → frame 3
succfr → frame 3

frame 5
succip → ③′
redoip → ④′
prevfr → frame 4
succfr → frame 2

frame 6
succip → ②′
redoip → do-fail
prevfr → frame 5
succfr → frame 5

frame 7
succip → ③
redoip → ②
prevfr → frame 6
succfr → frame 6

frame 8
succip → ③′
redoip → ④′
prevfr → frame 7
succfr → frame 5

frame 9
succip → ②′
redoip → ②
prevfr → frame 8
succfr → frame 8

frame 10
succip → ③′
redoip → do-fail
prevfr → frame 9
succfr → frame 8

Figura 7.27: *nondet stack* ao término da execução

ter que declarar os *modos* de uso de todos os predicados do programa, além de permitir que as descrições *SOS* possam ser mapeadas de forma simples para a linguagem *PAN* através do uso da notação *mixfix*.

7.8.1 Contribuições

- Implementação eficiente de uma linguagem lógica com os recursos descritos no capítulo anterior.
- Analisador interprocedural de *modos*.
- Analisador interprocedural de determinismo e uso de analisador de terminação para refinar resultados.
- Adaptação do modelo de execução proposto para linguagem *Mercury* com o intuito de suportar predicados de alta ordem e eliminar as restrições impostas (i.e. unificações entre variáveis livres).

Capítulo 8

Implementação do "framework"

8.1 Introdução

Em nosso "framework", o usuário precisa apenas especificar a semântica da linguagem e as aproximações desejadas. A partir destas especificações o usuário pode gerar analisadores, verificadores, interpretadores e depuradores de código. É lógico que a semântica utilizada para gerar um interpretador não é exatamente igual a utilizada para construir um analisador. Porém, como foi descrito no Capítulo 2, existe uma relação de "segurança" entre estas duas semânticas. É função do usuário garantir a existência desta relação.

É importante lembrar, que nem sempre é necessário definirmos a semântica da linguagem e as aproximações desejadas para obtermos uma ferramenta de análise e/ou verificação. Como foi descrito na Seção 5.3, em nosso "framework" estimulamos o uso de meta-linguagens (linguagens intermediárias). Assim, um usuário pode gerar ferramentas para uma linguagem X, simplesmente implementando um transformador da linguagem X para uma meta-linguagem já codificada dentro de nosso "framework". Portanto, neste capítulo, mostraremos como *PAN* pode ser utilizada como uma linguagem de transformação.

Neste capítulo apresentaremos as estruturas de dados e como os algoritmos descritos nos capítulos anteriores foram implementados em nosso "framework". Alguns destes estão implementados em C, com o intuito de melhorar a eficiência das ferramentas. De uma forma aproximada, podemos considerar que uma ferramenta é gerada a partir de uma descrição semântica e a seleção de um particular algoritmo de exploração do mapa de estados.

Como estamos utilizando uma abordagem modular para a especificação da semântica das linguagens de programação e especificação, podemos assumir que os módulos que compõe estas especificações também fazem parte do "framework". Visto que, estes podem ser reutilizados na descrição de novas linguagens.

No Capítulo 2 vimos que analisadores, em geral, substituem os domínios semânticos concretos por domínios semânticos aproximados. Para facilitar a construção destes analisadores, definimos uma pequena biblioteca de domínios semânticos aproximados. Esta biblioteca é composta por domínios, tais como o domínio de sinais e o domínio de intervalos. É importante ressaltar, que estes domínios aproximados também podem ser utilizados em processos de verificação, como descrito no Capítulo 5.

Desta maneira, os principais componentes de nosso "framework" são:

Linguagem *PAN* é utilizada principalmente para especificar a semântica, aproximada ou não, das linguagens de programação. *PAN* também, é utilizada para realizar transformações.

Estruturas de dados são utilizadas no processo de exploração do mapa de estados.

Algoritmos definem a técnica de exploração do mapa de estados.

Módulos semânticos são utilizados na definição da semântica, aproximada ou não, de uma linguagem de programação.

Domínios aproximados substituem os domínios concretos em processos de análise e verificação.

A seguir descreveremos detalhes sobre a implementação das estruturas de dados e dos algoritmos. Também mostraremos como utilizar *PAN* como um sistema transformacional. No próximo capítulo e no Apêndice E o leitor pode encontrar exemplos de módulos semânticos.

8.2 Estruturas de Dados

8.2.1 *Traceability*

Verificadores de programas devem produzir mensagens de erro com referências ao código do programa. Estas referências podem inclusive conter o número da linha de código que contém o erro. Contudo, verificadores trabalham sobre representações intermediárias que não possuem este tipo de informação. Para sanarmos este problema, o nosso "framework" fornece *mappings* que podem ser utilizados para armazenar informação adicional aos nodos de uma árvore de sintaxe abstrata.

Os *mappings* podem ser utilizados para associar informação de número de linha e coluna, e nome de arquivo a um determinado nodo da árvore de sintaxe abstrata. Esta associação é realizada pelo módulo de *análise sintática*, visto que este é o único módulo que possui acesso a este tipo de informação. A informação armazenada pode, então, ser acessada pelos módulos de análise e verificação.

Os *mapping* podem, também, ser utilizados para associar um nodo da representação intermediária a um nodo da árvore de sintaxe abstrata. Esta associação é realizada pelos módulos que transformam a árvore de sintaxe abstrata em uma representação intermediária (meta-linguagem).

Suponha o caso onde tenha sido detectado um erro e este esteja associado a um nodo N_{ir} da representação intermediária (meta-linguagem). O verificador pode, então, produzir uma mensagem de erro contendo informação de número de linha e coluna, e nome de arquivo. Para isto, este deve seguir a seqüência de associações $N_{ir} \leftrightarrow N_{as} \leftrightarrow Info$, onde N_{as} é um nodo da árvore de sintaxe abstrata e *Info* é a informação desejada.

Estes *mappings* são implementados em *C*, utilizando *hashtables*. Os *mappings* podem ser utilizados a partir de código *PAN* e *C*. A Figura 8.1 contém parte da interface do módulo de *mappings*.

8.2.2 Hashtable

Diversos analisadores e verificadores de código apresentados nos capítulos anteriores armazenam o mapa de estados em um estrutura de dados. Por razões de eficiência, uma *hashtable* é, em geral, utilizada. Para obtermos eficiência, o nosso "framework" provê uma implementação de *hashtables* em *C*. A Figura 8.2 contém parte da interface do módulo de *hashtable*.

8.2.3 Array de bits

O método *Supertrace* utiliza um *array* de bits que não pode ser implementado de forma eficiente em *PAN*. Por isso, o nosso "framework" possui uma implementação desta estrutura de dados em *C*. A *interface* é semelhante as *interfaces* dos *mappings* e *hashtables*.

```
cdecl create-mapping : in (in A -> out int -> o) ->
                       in (in A -> in A -> o) ->
                       unq out mapping A B -> o {det}
C{{ ... }}C.

cdecl add-mapping-entry : in A -> in B ->
                          unq in mapping A B ->
                          unq out mapping A B -> o {det}
C{{ ... }}C.

cdecl remove-mapping-entry : in A -> in B ->
                             unq in mapping A B ->
                             unq out mapping A B -> o {det}
C{{ ... }}C.

cdecl is-mapped-to : in A -> in mapping A B ->
                     out list B -> o {det}
C{{ ... }}C.
```

Figura 8.1: *Mapping Interface*

```
cdecl create-hashtable : in (in A -> out int -> o) ->
                         in (in A -> in A -> o) ->
                         unq out hashtable A -> o {det}
C{{ ... }}C.

cdecl add-hashtable-entry : in A -> unq in hashtable A ->
                            unq out hashtable A -> o {det}
C{{ ... }}C.

cdecl get-hashtable-entry :  in A -> in hashtable A ->
                             out A -> o {det}
C{{ ... }}C.

cdecl remove-hashtable-entry : in A -> unq in hashtable A ->
                               unq out hashtable A ->
                               o {det}
C{{ ... }}C.
```

Figura 8.2: *Hashtable Interface*

8.2.4 *Trace* corrente

Ao contrário das estruturas de dados descritas nas seções anteriores, o *trace* corrente não precisa ser codificado em *C*. A implementação desta estrutura de dados se limita ao uso de listas.

8.3 Algoritmos

Todos os algoritmos descritos nesta seção assumem que o usuário tenha definido um predicado de transição da seguinte forma:

```
type _ -- _ --> : config -> label -> config -> o.
```

É importante ressaltar que não estamos privilegiando um estilo de especificação. Todos, os estilos de especificação *SOS* apresentados no Capítulo 4 se encaixam nesta definição. Até mesmo especificações que utilizem *instruction pointers* no lugar de termos podem ser utilizadas.

Além do predicado de transição, o usuário também deve fornecer:

- predicados para extração do estado antes e depois da transição.

```
type get-pre-state : in config -> in label ->
                                in config -> out state.
type get-post-state : in config -> in label ->
                                 in config -> out state.
```

- predicado para comparar estados.

```
type compare-states : in state -> in state -> o.
```

Todos os predicados auxiliares descritos acima podem ser implementados usado a linguagem *C*.

8.3.1 Interpretador ou Simulador

Um interpretador ou simulador é definido através do fecho transitivo do predicado de transição. Em nosso "framework" existem diversas variações deste algoritmo. O mais simples de todos simplesmente executa o programa para uma determinada entrada, produzindo uma saída. Além da definição do predicado de transição, este interpretador assume a existência dos predicados contidos na Figura 8.3. O predicado `completed` verifica se uma configuração é terminal, isto é, se esta representa o final da execução do programa. O predicado `id` define o *label* identidade. O predicado `compose` define como é realizada a composição dos *labels*. O predicado `set-label-input` inicializa um *label* com os dados de entrada, e `get-label-ouput` com os dados de saída. A Figura 8.4 contém a definição do predicado `execute` que implementa o interpretador. O predicaod `_ -- _ -->> _` define uma relação entre as configurações iniciais e finais, isto é, entre a entrada e o resultado de um programa.

Este interpretador simples é útil para a realização de simulações em linguagens determinísticas. Se o predicado `_ -- _ --> _` for não determinístico, este interpretador é de pouco uso, visto que o usuário não possui controle sobre a seleção do *trace* de execução. A partir deste problema, criamos diversas variações deste interpretador. Estas variações utilizam o "truque" de obter todas as soluções do *goal* `C1 -- L1 --> C1'`, utilizando o predicado *built-in* `solutions`.

```
type completed : config -> o.
type id : label -> o.
type compose : label -> label -> label -> o.
type set-label-input : input -> label -> o.
type get-label-ouput : label -> output -> o.
```

Figura 8.3: Predicados utilizados pelo interpretador

```
type _ -- _ -->> _ : config -> label -> config -> o.

C -- Ln -->> C :- completed C, id Ln.

C1 -- Ln -->> Cn :- compose L1 Ln' Ln,
                    C1 -- L1 --> C1',
                    C1' -- Ln' -->> Cn'.

entry execute : in config -> in input -> out output -> o.
execute C0 In Out :- set-label-input In L,
                     C0 -- L -->> Cn,
                     get-label-ouput L Out.
```

Figura 8.4: Definição do interpretador

As soluções deste *goal* representam todas as possíveis transições que um programa pode realizar em um determinado estado. De posse de todas transições possíveis, o interpretador pode requisitar que o usuário escolha uma. Outra variação consiste em o usuário fornecer um predicado "oráculo" que realiza a escolha. Este predicado "oráculo" pode, por exemplo, estar utilizando um gerador de números aleatórios para realizar a escolha.

Para auxiliar o processo de simulação, o usuário pode utilizar *breakpoints* para interromper o processo de interpretação em um determinado ponto ou quando uma condição específica for satisfeita.

A técnica de verificação "execução livre" pode ser implementada através dos predicados "oráculo". Este predicado seleciona um particular *trace* de execução. Caso nenhum erro seja detectado no *trace* escolhido após um número X de passos, o processo é reinicializado utilizando um outro *trace*. Este processo de reinicialização é igual ao utilizado na ferramenta *Verisoft* [39].

8.3.2 Verificadores

Nesta seção apresentaremos apenas os verificadores que não constroem o mapa de estados explicitamente. É importante lembrar que a função dos verificadores é detectar erros. Verificadores que constroem o mapa de estados explicitamente podem ser implementados usando as técnicas para a construção de analisadores.

Para construirmos um verificador, simplesmente armazenamos informações sobre o *trace* corrente dentro do *label* do predicado de transição. A informação armazenada depende do tipo de função de detecção de divergência que será utilizada. Por exemplo, se o usuário deseja utilizar uma função de detecção divergência que interrompe o processamento após n passos, então o *label* deve conter o número de passos no *trace* corrente. Se o usuário deseja utilizar uma função de detecção de divergência que interrompe o processo após a identificação de n *hashcodes* repetidos, então uma lista contendo os *hashcodes* dos estados do *trace* corrente deve ser armazenada dentro do *label*. Analogamente, para o caso a função de detecção divergência baseada no "peso" dos estados (Seção 5.6.2.3). Note que até os próprios estados do *trace* corrente podem ser armazenados no *label*, com o intuito de produzir uma função de detecção de divergência mais sofisticada [1].

Obviamente, no caso das funções de detecção de divergência baseadas em *hashcodes* e "pesos", o usuário deve fornecer os seguintes predicados:

```
type get-hash : in state -> out int -> o.
type get-weight : in state -> out int -> o.
```

O nosso "framework" fornece diversos predicados que implementam as funções de detecção de divergência descritas na Seção 5.6.2.3. Mas nada impede que o usuário defina os seus próprios predicados.

O predicado de detecção de divergência é utilizado para decidir se o *trace* corrente deve ou não ser abandonado. Este processo pode ser implementado, de forma simples, através do predicado *built-in* `fail`. Considere o seguinte trecho de código:

```
C -- L --> C' :- checkDivergence L, !, fail.
```

Quando o predicado `checkDivergence` tem sucesso, então, o predicado `fail` encarrega-se de descartar o *trace* corrente. Os mecanismos de *backtracking* da linguagem *PAN* faz com que o próximo *trace* passe a ser analisado.

Caso um *trace* termine, o verificador deve passar a executar uma nova alternativa. Mais uma vez, isto é codificado através do `fail`.

[1] E na maioria dos casos menos eficiente.

```
C -- L --> C' :- completed C, !,
                 fail.
```

Esta estrutura de implementação reflete o nosso interesse em detectar erros, já que o predicado `_ -- _ --> _` tem sucesso apenas no caso em que um erro é detectado.

Como já mencionado, o algoritmo de *supertrace* (Seções 3.5.2 e 5.6.2.3) pode ser encaixado na implementação descrita acima. Entretanto, neste caso, o *label* não contem apenas os *hashcodes* do *trace* corrente, mas sim um *array de bits* contendo informação sobre todos os *hashcodes* de estados já visitados.

Obviamente, esta estrutura simples de implementação permite apenas a construção de verificadores de código que fazem buscas em profundidade nos mapas de estado. Acreditamos que esta seja a maneira natural de construir este tipo de ferramenta, que é compartilhada por sistemas como o *SPIN* [47]. Entretanto, é possível realizar buscas em largura, mas neste caso será necessário utilizar o predicado *built-in solutions* de forma semelhante a descrita na seção anterior.

Os erros são detectados utilizando os algoritmos descritos no Capítulo 5. As marcações *assert*, *assure*, *Assert* e *Assure* são triviais de serem implementadas, uma vez que estas checam apenas informação local. As marcações do tipo *assert* geram uma mensagem de erro e terminam a execução do programa, quando as condições específicas nestas não são satisfeitas. Por outro lado, as marcações do tipo *assure* são implementadas de forma semelhante as funções de detecção de divergência. Quando a condição especificada em uma marcação *assure* não é satisfeita, então, esta utiliza o predicado `fail` para descartar o *trace* corrente (Seção 5.6.2.5).

A verificação de propriedades temporais é um pouco mais complexa, visto que é necessário modificar a estrutura dos *labels* e configurações. Lembre que para verificar uma propriedade temporal é necessário associarmos um estado do programa a um estado do sistema de transição $\mathcal{ERROR}$ que representa a situação de erro (Seção 5.6.2.4). Isto nada mais é do que definir um par ordenado $\langle s, e \rangle$, onde s é o estado do programa, e e o estado do sistema de transição $\mathcal{ERROR}$. Nos nossos protótipos, o sistema de transição $\mathcal{ERROR}$ é descrito utilizando uma linguagem textual convencional. Infelizmente, não implementamos um programa que converta propriedades descritas em lógica temporal linear em sistemas de transição $\mathcal{ERROR}$. Logo, esta conversão deve ser realizada manualmente pelo usuário.

Em nossos protótipos, o sistema de transição $\mathcal{ERROR}$ nada mais é do que um processo que roda em paralelo com o programa a ser verificado (Seção 5.6.2.4). Note que esta abordagem só pode ser implementada em linguagens que possuam suporte a criação de processos. Se o usuário quiser realizar verificações temporais em linguagens que não possuam suporte a paralelismo, ele/ela terá que adicionar paralelismo a semântica da linguagem. No próximo capítulo veremos que isto não é complicado, uma vez que estamos trabalhando com descrições modulares.

8.3.3 Analisadores

Os analisadores são codificados mapeando os algoritmos descritos no Capítulo 2 para a linguagem *C*. Como a semântica (aproximada) da linguagem está codificada em *PAN*, é necessário utilizar a "interface *PAN*/*C*" para implementar estes algoritmos. Abstratamente, a descrição semântica, codificiada em *PAN*, "implementa" a função *successors* utilizada nos algoritmos descritos no Capítulo 2.

A implementação destes algoritmos também utiliza as estruturas de dados descritas no início deste capítulo. Estas estruturas são utilizadas para armazenar o grafo contendo o mapa de estados. Note que, neste caso não é necessário a utilização da interface entre as linguagens *PAN* e *C*, visto que os algoritmos e as estruturas de dados estão implementados em *C*.

A implementação do algoritmo de análise interprocedural descrito na Seção 5.6.3.1 é mais complexa. Como este algoritmo depende da semântica da linguagem (mecanismo de passagem de parâmetros, modo de avaliação, etc), não foi possível (ainda) definir um algoritmo genérico (reutilizável). Entretanto, implementamos versões específicas deste algoritmo para algumas das linguagens exemplo codificadas dentro do *framework*.

8.4 PAN como sistema de transformação

Como já mencionado na Seção 5.3, em nosso "framework" estimulamos o uso de meta-linguagens. Através do uso de meta-linguagens podemos simplificar em muito a definição de analisadores.

Para especificarmos a transformação de programas de uma linguagem em programas de uma meta-linguagem, ou de programas de uma meta-linguagem em programas de outra meta-linguagem, utilizamos a linguagem *PAN*. Esta decisão é justificada na literatura, onde *Prolog* já foi utilizado com sucesso na definição de transformações [94], implementação de compiladores [102] e gramáticas de atributos [90].

Neste caso, estamos usando *PAN* como um sistema de transformação, onde a unificação funciona como um mecanismo de casamento de padrões.

Ao contrário de ambientes como o *Draco* [69, 70, 58] e *DMS* [9], as transformações são realizadas sobre uma estrutura abstrata de dados (tipo especificado em *PAN*), e não sobre a árvore gerada pelo analisador sintático. A nossa abordagem tem a desvantagem de obrigar o usuário a construir o termo da estrutura abstrata de dados que representa o programa. Ou seja, o usuário tem que colocar ações para construir o termo desejado dentro das regras de redução. Por outro lado, a nossa abordagem tem a vantagem de desacoplar a estrutura de dados usada no processo de transformação da técnica de análise sintática utilizada. Tanto, no sistema *Draco*, quanto no sistema *DMS*, a árvore de sintaxe abstrata gerada é uma função direta da forma como a gramática da linguagem foi especificada. Do ponto de vista de engenharia de software, isto é um grande problema, visto que a modificação da gramática (por motivos de eficiência), obriga o usuário a modificar as regras de transformação. Outro problema, é que para conseguirmos definir uma gramática *LaLR(1)* muitas vezes são utilizados artifícios que acabam por produzir árvores nem um pouco intuitivas, que complicam a definição das regras de transformação.

De forma geral, para implementarmos um transformador, devemos definir um predicado com a seguinte estrutura:

```
type _ == _ ==> _ : ast1 -> context -> ast2 -> o.
```

Onde o tipo *ast1* é o tipo da árvore de sintaxe abstrata "fonte", *context* é o contexto de transformação, e *ast2* é o tipo da árvore de sintaxe abstrata "destino". O contexto de transformação pode ser utilizado para passar parâmetros adicionais para uma transformação, ou até mesmo informações sobre o contexto global. Da mesma forma, que produzimos descrições semânticas modulares, também podemos obter transformadores modulares, se a estrutura do contexto de transformação estiver encapsulada. A seguir descrevemos alguns exemplos de transformação.

```
(V := Expr) == Context ==> (ExprRes, assign V Tmp) :-
    Expr == Context ==> ExprRes,
    get-tmp-var-with-result Context Tmp.
```

O exemplo acima é um fragmento de um transformador que converte uma linguagem imperativa simples em uma representação intermediária. Este transformador foi utilizado num dos exemplos apresentados no próximo capítulo. Para transformar uma atribuição, a expressão é transformada, e em seguida o "nome" da variável temporária contendo o resultado da expressão

é obtida através do predicado *get-tmp-var-with-result*. Esta informação encontra-se dentro do contexto de transformação.

```
if Expr then Cmd1 else Cmd2 == Context ==> Cmd1' :-
       static-eval Context Expr true,
       Cmd1 == Context ==> Cmd1'.
```

No exemplo acima, o contexto de transformação contem informações obtidas durante a análise do programa. Estas informações são utilizadas para verificar se o valor de uma expressão pode ser estaticamente determinado. Note que o transformador é invocado recursivamente com o intuito de transformar o corpo do *if*.

O próximo e último exemplo mostra como codificar equações semânticas de semântica denotacional ou semântica de ações, em *PAN*. Lembre que estas equações semânticas podem ser vistas como um tipo de transformação. Desta forma, as equações semânticas:

execute $[\![C_1 \ ; \ C_2]\!]$ = execute C_1 and then execute C_2.

elaborate $[\![$ var $I : T]\!]$ =
  | allocate a cell
  then
  | bind I to the given cell

podem ser codificadas em *PAN* como:

```
type execute [[ _ ]]= _ : Command -> Action -> o.
type elaborate [[ _ ]]= _ : Declaration -> Action -> o.

execute [[ C1 ; C2 ]]= ExecuteC1 and then ExecuteC2 :-
                 execute [[ C1 ]]= ExecuteC1,
                 execute [[ C2 ]]= ExecuteC2.

elaborate [[ var I : T ]]= (allocate a cell)
                           then
                           (bind I to the given cell).
```

8.5 Conclusão

Neste capítulo, apresentamos como as técnicas de análise e verificação de programas foram implementadas em nosso "framework". Mostramos os principais componentes do "framework" e como estes podem ser utilizados. Este capítulo também deixa claro como é relativamente simples implementar este tipo de ferramenta.

Note que as ferramentas de análise e verificação podem ser implementadas em outras linguagens. Entretanto, é importante lembrar que os mecanismos de *backtracking* e as facilidades para a descrição de programas não determinísticos existentes na linguagem *PAN* facilitam muito a implementação.

Capítulo 9

Exemplos de Analisadores e Verificadores

9.1 Introdução

Neste capítulo apresentamos exemplos de analisadores e verificadores de código. Os primeiros exemplos são realizados sobre uma linguagem imperativa simples. Em seguida, mostramos exemplos envolvendo linguagens concorrentes, alocação dinâmica de memória e DSLs. O objetivo deste capítulo é mostrar como o "framework" é utilizado e apresentar técnicas de análise e verificação para lidarmos com: alocação dinâmica de memória, alocação dinâmica de processos, etc. Servindo desta forma, como "fonte de idéias" para a implementação de novos analisadores e verificadores com o nosso "framework".

Deve ficar claro, que este capítulo não pretende ser exaustivo e apresentar todos os possíveis tipos de ferramentas de análise e verificação apresentados nos capítulos anteriores. O leitor interessado pode consultar o Apêndice E onde encontrará uma especificação de tamanho considerável escrita em *PAN*. Esta especificação é a codificação da semântica de algumas facetas da *notação de ações* utilizada no formalismo de *semântica de ações* [66].

9.2 Linguagem Imperativa Simples

O objetivo desta linguagem é apresentar didaticamente como interpretadores e analisadores de código são construídos em nosso "framework". Inicialmente, apresentamos a especificação em *PAN* da semântica operacional da linguagem. A partir desta especificação é possível realizar simulações (interpretar programas).

Na próxima seção apresentaremos uma linguagem (representação) intermediária para a linguagem descrita nesta seção. A semântica da linguagem intermediária possui um nível de abstração mais baixo, utilizando, por exemplo, *instruction pointers*. Esta semântica de mais "baixo nível" é o ponto de partida para a implementação dos analisadores de sinal, intervalo e propagação de constantes.

A Figura 9.1 contém um fragmento da especificação da árvore de sintaxe abstrata da linguagem imperativa. A Figura 9.2 contem um fragmento da especificação da semântica da linguagem. O *label* está encapsulando o *estado da computação* antes e depois da transição. O predicado *set-store* é utilizado para modificar o valor associado a um determinado identificador. O predicado *is-val* é utilizado para verificar se uma expressão é um valor atômico.

```
kind stmt : type.
kind expr : type.

type _ ; _ : stmt -> stmt -> stmt {prec 80 l-assoc}.
type skip : stmt.
type if _ then _ else _ end :
                expr -> stmt -> stmt -> stmt {prec 65}.
type while _ do _ end : expr -> stmt -> stmt {prec 70}.
type _ := _ : id -> expr -> stmt {prec 100}.
```

Figura 9.1: Fragmento da especificação da sintaxe abstrata

```
kind label : type.
kind config : type.
sub stmt : config.
sub expr : config.
type completed : config.
type _ -- _ --> _ : config -> label -> config -> o.

S1 -- L --> S1'
|-
S1 ; S2 -- L --> S1' ; S2.

id L
|-
while E do S end -- L --> if E then S ; (while E do S end) end.

(is-val E V), id L', (set-store L' ID V L)
|-
ID := E -- L --> completed.

E -- L --> E'
|-
if E then S else S' -- L --> if E' then S else S'.

id L
|-
if true then S else S' -- L --> S.

id L
|-
if false then S else S' -- L --> S'
```

Figura 9.2: Fragmento da semântica da linguagem

```
kind opcode.
type assign : qid -> qid   -> opcode.
type set    : qid -> value -> opcode.
type lbl    : qid -> opcode.
type goto   : qid -> opcode.
type bz     : qid -> qid -> opcode.
type add    : qid -> qid -> qid -> opcode.
```

Figura 9.3: Fragmento da especificação da sintaxe abstrata

9.2.1 Linguagem de "Baixo Nível"

Como já mencionado, apesar de ser possível construir analisadores a partir de descrições (modulares ou não) SOS "convencionais", o analisador não terá uma eficiência satisfatória. O problema é que em descrições SOS "convencionais" o termo que representa o programa faz parte do estado do mesmo. Como estes termos são em geral muito grandes, o estado do programa, conseqüentemente, ficaria muito grande. Logo, operações de comparação e cópia de estados utilizadas pelos algoritmos de análise apresentados nos Capítulos 2 e 5 tornariam-se muito ineficientes. Deve ficar claro para o leitor, que no caso de DSLs este problema pode não ser tão grave, visto que, programas (especificações) codificadas neste tipo de linguagem tendem a ser mais compactos.

Por estes motivos, nesta seção apresentamos uma linguagem intermediária de "baixo nível". Programas escritos na linguagem imperativa descrita na seção anterior podem ser transformados em programas da linguagem intermediária. Note que, esta prática é muito comum na implementação dos *backends* de compiladores "convencionais", onde o programa é convertido para uma representação intermediária, no qual é aplicada otimizações. É importante lembrar, que *PAN* pode ser utilizada para converter programas escritos na linguagem imperativa para programas da linguagem intermediária.

A linguagem intermediária de baixo nível possui os seguintes *opcodes:*

assign *src dest* : Atribui o valor da variável *src* a variável *dest*.

set *dest val* : "Seta" o valor da variável *dest* com o valor *val*.

lbl *l* : Indica uma localização no programa.

goto *l* : "Salta" para a localização especificada.

bz *lbl src* : "Salta" para a localização especificada, se o valor da variável *src* for zero.

***opr src1 src2 dest* :** Realiza uma operação binária, *opr* pode ser: *add*, *sub*, etc.

A Figura 9.3 contem parte da especificação destes *opcodes* em *PAN*. Note que, *qid* (*quoted id*) é um tipo *built-in* da linguagem *PAN*, que pode ser utilizado para criar identificadores. A seguir mostramos alguns exemplos de "*quoted ids*".

- 'x
- 'var
- 'tst

```
type get-ip : in state -> out ip -> o.
type get-opcode : in label -> in ip -> out opcode -> o.
type get-store : in state -> in qid -> out value -> o.
type set-store : in state -> in qid -> in value -> out state -> o.
type next      : in label -> in state -> out state -> o.
type get-lbl-ip  : in label -> in qid -> out ip -> o.
type set-ip : in state -> in ip -> out state -> o.
type add-op : in value -> in value -> out value -> o.
```

Figura 9.4: Assinatura de alguns predicados utilizados na descrição semântica

A Figura 9.4 contem a assinatura de alguns predicados utilizados na especificação da linguagem intermediária. Utilizamos os marcadores *in* e *out* apenas para indicar ao leitor quais predicados são de entrada e quais são de saída. Antes de explicarmos a semântica de cada um destes predicados, devemos definir a estrutura das configuração e do *label* utilizados. Na linguagem imperativa descrita na seção anterior a *configuração* era um termo ou valor e o *label* encapsulava o estado antes e depois da transição. Neste exemplo, a configuração será o *estado* da computação. O *estado* da computação é composto pelo *instruction pointer* e pelo *store*. O *instruction pointer* nada mais é do que um índice que indica o *opcode* corrente. Um programa escrito na linguagem intermediária nada mais é do que uma seqüência de *opcodes*. Neste exemplo, a seqüência esta encapsulada dentro do *label*, já que precisamos da seqüência para acessarmos o *opcode* corrente.

A lista a seguir contem uma descrição breve dos predicados contidos na Figura 9.4.

get-ip : "pega" o *ip* corrente.

get-opcode : "pega" o *opcode* associado a um dado *ip*.

get-store : "pega" o valor associado a um identificador dentro do *store*.

set-store : modifica o valor associado a um identificador dentro do *store*.

next : "avança" (se possível) o *ip* corrente, i.e. vai para o próximo *ip*.

get-lbl-ip : "pega" o *ip* associado a um determinado *lbl* (rótulo do programa).

set-ip : modifica o valor do *ip* corrente.

add-op : adiciona dois "valores".

A Figura 9.5 contem um fragmento da semântica da linguagem intermediária. Apesar desta "descrição semântica" não seguir o estilo *SOS*, o leitor pode verificar que esta é uma descrição modular, onde a estrutura do *estado da computação* e do *label* estão encapsulados. Adiante mostraremos como reutilizar esta descrição na definição de uma linguagem paralela.

9.3 Analisadores

Implementar analisadores de código a partir da representação intermediária apresentada na seção anterior é bastante simples. O primeiro passo consiste em substituir o tipo *value* por uma versão abstrata. Os operadores (predicados) relativos ao tipo *value* também devem ser modificados.

```
// predicados auxiliares
type op : label -> state -> opcode -> o.
type jump : label -> state -> lbl -> state -> o.
jump L ST LBL ST' :- get-lbl-ip L LBL IP, set-ip ST IP ST'.
op L ST OP :- get-ip ST IP, get-opcode L IP OP.

ST -- L --> ST'  :-  op L ST (assign SRC DEST),
                     get-store ST SRC V,
                     set-store ST DEST V STtmp,
                     next L STtmp ST'.

ST -- L --> ST'  :-  op L ST (set DEST V),
                     set-store ST DEST V STtmp,
                     next L STtmp ST'.

ST -- L --> ST' :-   op L ST (label LBL),
                     next L ST ST'.

ST -- L --> ST' :-   op L ST (goto LBL),
                     jump L ST LBL ST'.

ST -- L --> ST' :-   op L ST (bz LBL SRC),
                     get-store ST SRC V,
                     if (V = 0)
                     then (jump L ST LBL ST')
                     else (next L ST ST').

ST -- L --> ST' :-   op L ST (add SRC1 SRC2 DEST),
                     get-store ST SRC1 V1,
                     get-store ST SRC2 V2,
                     add-op V1 V2 V
                     set-store ST DEST V STtmp,
                     next L STtmp STtmp'.
```

Figura 9.5: Fragmento da semântica da linguagem intermediária

Por exemplo, no caso de um analisador de sinais, o predicado *add-op* deve ser substituído por uma versão abstrata que se comporta como o operador de soma abstrato descrito no Capítulo 2.

Algumas transições também devem ser modificadas/adicionadas, visto que o uso de aproximações faz com que alguns *opcodes* tornem-se não determinísticos. No caso de um analisador de sinal, as transições relativas ao *opcode bz* devem ser modificadas. Suponha que nosso reticulado de sinais contenha apenas os elementos: *pos*, *neg*, *zero* e $\top$ (top). Então a transição associada ao *opcode bz* ficará da seguinte forma:

```
ST -- L --> ST' :-
 op L ST (bz LBL SRC),
 get-store ST SRC V,
 (   (V = zero, jump L ST LBL ST')
   ; (V = top, (jump L ST LBL ST' ; next L ST ST'))
   ; (V = pos, next L ST ST')
   ; (V = neg, next L ST ST')
 )
```

Isto é, se $V = 0$ o programa realiza o salto; se $V = pos$ ou $V = neg$ passa a executar o próximo *opcode*; se $V = top$, então o programa pode saltar *ou* continuar a execução no próximo *opcode*.

O nosso protótipo contem a implementação de um analisador de sinais, intervalo e propagador de constantes para a linguagem intermediária.

O diagrama de fluxo de dados de um programa escrito na linguagem intermediária pode ser facilmente extraído. Precisamos apenas substituir o tipo *value* por um tipo contendo apenas um valor, isto é, contendo apenas *top*. Neste caso, o mapa de estados produzido ser o diagrama de fluxo de controle do programa.

9.4 Alocação dinâmica de memória

A análise de linguagens que suportam alocação dinâmica de memória é sempre problemática, uma vez que uma quantidade ilimitada de memória pode ser alocada. Por exemplo, um programa simples como o definido a seguir possui uma quantidade potencialmente infinita de estados.

```
while (true)
        malloc(100);
```

Logo, algum tipo de aproximação deve ser utilizada para garantir a terminação dos analisadores. Também é necessário definir algum tipo de representação para as localizações abstratas de memória.

A nossa solução para este "problema" usa uma abordagem semelhante a utilizada para tratar funções recursivas (Seção 5.6.3.1). Assim, uma localização abstrata de memória é identificada pelo contexto de uso da mesma. Se este contexto não for aproximado, então cada localização da memória é precisamente identificada. Obviamente, este contexto não garantirá o término da análise.

Para garantir a terminação, podemos utilizar um contexto aproximado que consiste apenas da posição do programa onde a memória foi alocada. Logo, no programa descrito acima, todas os blocos de memória alocados dentro do *while* seriam "fundidos" em uma única localização abstrata de memória.

Infelizmente, esta abordagem "simples" tem alguns problemas, pois é muito comum encontrarmos programas que possuam a sua própria função de alocação de memória, que por sua vez invoca a função (ou primitiva) de alocação de memória da linguagem. Por exemplo, considere o programa a seguir:

```
void * my_malloc (size nbytes)
{
  ...
  tmp = malloc(nbytes + aux);
  ...
}
```

Neste exemplo, a função *my_malloc* definida pelo usuário é utilizada para fazer todas as alocações de memória dentro do programa. Portanto, todos os blocos de memória alocados dentro do programa seriam "fundidos" durante a análise, desde que exista apenas um lugar no programa que invoque a função *malloc*.

Felizmente, este "problema" pode ser resolvido facilmente, basta usarmos uma abordagem semelhante a *nCFA* descrita na Seção 5.6.3.1. Neste caso, o contexto abstrato é representado pelas n últimas funções contidas na pilha de execução, e a posição do programa onde o *malloc* foi realizado.

Deve estar claro que outras alternativas podem ser utilizadas. Por exemplo, podemos adicionar um *id* a definição do contexto abstrato. A função deste *id* é evitar que as localizações de memória diferentes sejam "fundidas". Obviamente, não é possível gerar um *id* diferente para cada localização de memória, já que neste caso o problema de terminação apareceria novamente. Entretanto, podemos utilizar um conjunto finito de tamanho m de *ids*. Assim o analisador conseguiria diferenciar até m localizações de memória que foram alocadas em um mesmo contexto abstrato. É importante lembrar que quanto maior for m, maior será o mapa de estados do programa, e quanto maior for m, mais precisa será a análise.

A semântica da atribuição deve também ser modificada para lidar com localizações abstratas de memória. Basicamente, temos que diferenciar localizações (abstrata) de memória que tenham ou não sido "fundidas". Uma localização de memória que não tenha sido "fundida" é tratada normalmente. Entretanto, atribuições a posições de memória que tenham sido "fundidas" devem ser tratadas de forma "incremental", isto é:

$$\textit{valor-após-atualização} = \textit{valor-novo} \sqcup \textit{valor-antigo}$$

Para exemplificar a semântica da atribuição na presença de localizações abstratas de memória "fundidas", considere o programa a seguir:

```
for(i = 0; i < 100; i++)
{
  int * data = malloc(sizeof(int));
  *data = i;
}
```

Este programa aloca 100 localizações de memória e as inicializa com os valores 0, 1, ... e 99, respectivamente. Mas, o analisador"funde" estas 100 localizações de memória em uma única localização abstrata e no final da análise esta localização conterá o intervalo $[0, 99]$, se inteiros estiverem sendo aproximados por intervalos. Obviamente, esta é uma aproximação segura do efeito do programa.

9.5 Exemplos de Verificação

Nesta seção, em vez de descrevermos exemplos conhecidos relacionados a linguagens concorrentes, focamos em aplicações não usuais das técnicas de verificação descritas nos capítulos

```
string arrayCat(string [ ] a)
{
  wr = new StringWriter();
  for (i = 1; i <= size(a); i++)
    putString(wr, a[i]);
  return getString(wr);
}
```

Figura 9.6: Função *ArrayCat*

anteriores. Nossa intenção é ilustrar como estas técnicas podem ser utilizadas para detectar erros tais como índices inválidos de *arrays*, dereferenciação de ponteiros nulos, etc.

Figura 9.6 contem uma função simples que possui vários erros. O protótipo usado nestes exemplos foi baseado nas descrições semânticas apresentadas nas seções anteriores. Este exemplo é baseado em um outro usado para exemplificar o funcionamento do sistema ESC [32]. ESC utiliza uma abordagem baseada em prova automática de teoremas.

A função *arrayCat* concatena os elementos da *string a*. O tipo abstrato de dados *StringWriter* modela uma saída de dados do tipo *stream*. A função *putString* insere uma *string* no fim da *stream*. A função*getString* retorna o conteúdo da *stream*. Para verificar este programa, utilizaremos as seguintes aproximações:

1. *String* é aproximada pelos valores abstratos *null* e *non-null*.

2. *StringWriter* é aproximada pelos valores abstratos *uninit* e *init*, representando uma *StringWriter* antes e depois da inicialização.

Note que também é necessário fornecer versões abstratas dos operadores e funções que lidam com *Strings* e *StringWriters*.

Agora, podemos realizar verificações considerando os contextos de uso da função *arrayCat* em todo o programa. Uma possibilidade é escrever funções auxiliares que "artificialmente" produzem diferentes contextos de uso. De qualquer forma, a ferramenta de verificação detectará os seguintes erros:

1. A função *putString* é aplicada sobre um objeto não inicializado (*uninit*). O usuário pode consertar este problema adicionando o comando *initStringWriter(wr)*.

2. Uso inválido da função *putString*, a *string a[i]* pode ser *null*. O usuário pode consertar este problema adicionando um *if*:

   ```
   if (a[i] != null)
     putString(wr, a[i]);
   ```

3. Índice inválido de *array* na expressão *a[i]*. O usuário pode consertar este erro trocando

   ```
   for (i = 1; i <= size(a); i++)
   ```

   por

   ```
   for (i = 0; i <= size(a) - 1; i++)
   ```

```
int i;
char buf[1024];
file fd; // declara um file handler
...
fd = open(filename);
...
read(fd, buf, i); // lê i bytes, e copia estes para buf
...
close(fd);
...
read(fd, buf, i);
```

Figura 9.7: Programa de acesso a arquivo

A Figura 9.7 contem outro exemplo utilizado em nosso protótipo, neste caso, as seguintes aproximações são utilizadas:

1. valores inteiros(*int*) são aproximados por intervalos.
2. *file handlers* são aproximados pelos valores abstratos *closed* e *opened*.

Se o valor máximo do intervalo associado a variável i for maior do que 1023 no primeiro *read*, então um índice inválido de *array* é detectado. Se o erro for detectado devido ao uso de aproximações semânticas, então o usuário pode adicionar *assure(i < 1024)* para evitar a geração de mensagens de erro "falsas". A ferramenta de verificação também detectará um erro no último *read*, uma vez que o valor abstrato associado a *fd* é *closed*.

9.6 Verificadores e Aproximações

Na seção anterior, vimos exemplos de verificação que utilizam aproximações semânticas. Nestes exemplos, foi utilizada uma aproximação comum para todos os dados de um determinado tipo. Por exemplo, utilizar intervalos para aproximar todos os valores inteiros no exemplo da Figura 9.7. Infelizmente, nem sempre desejamos utilizar uma mesma aproximação para todos os dados de um determinado tipo. Para resolver este "problema", extendemos a linguagem descrita na seção anterior, para a mesma suportar o uso de marcações que possibilitem a especificação de aproximações.

Estas marcações passam a fazer parte do *design* do programa e não devem ser descartadas após o processo de verificação. Por este motivo, insistimos que esta deva fazer parte do código do programa. Como diferentes tipos de verificação implicam em aproximações (modelos) diferentes, introduzimos o conceito de contexto de verificação. Assim, temos marcações para a definição de contextos de verificação e marcações para a especificação de aproximações dentro de um contexto de verificação.

A Figura 9.8 contém um exemplo de uso dos contextos de verificação. Todos os comandos pertencentes a sublinguagem de aproximação estão entre os símbolos "*(**" e "**)*". O comando "*verification context*" define um novo contexto de verificação. O comando "*VC: use APROX*" indica que no contexto de verificação *VC* deve ser utilizado a aproximação *APROX* para a última variável declarada. O comando "*VC: begin ignore*" indica que o trecho de código até o comando "*VC: end ignore*" deve ser ignorado pelo contexto de verificação *VC*. O comando "*VC:*

```
(* verification context VC1 *)
(* verification context VC2 *)

int x; (* VC1: use DOT *)
       (* VC2: use SIGN *)
int y; (* VC1, VC2: use INTERVAL *)
(* VC1: begin ignore *)
...
// complex procedure 1
...
(* VC1: end ignore *)
while (...)
{
  ...
  x := x + 1;
  (* VC2: begin replace *)
  ...
  // complex procedure 2
  ...
  (* VC2: replaced by *)
  (* VC2: // simple abstraction of the complex procedure 2 *)
  (* VC2: ...*)
  (* VC2: end replace *)
}
```

Figura 9.8: Exemplo de uso dos contextos de verificação

begin replace" é semelhante ao *ignore*, a diferença é que o trecho de código é substituído por uma versão mais simples (abstrata).

Estas marcações também evitam práticas de verificação *ad-hoc* utilizadas no sistema *SPIN* [47]. Neste sistema de verificação é muito comum o usuário fazer "*copy&paste*" de trechos de código e em seguida editar o código manualmente para "especificar aproximações". Obviamente, esta técnica tem diversas desvantagens, dentre elas o problema de consistência entre as cópias criadas.

9.7 Linguagem Concorrente

Nesta seção apresentamos simuladores e analisadores de código para uma linguagem concorrente. Para facilitar a exposição, iremos começar com uma linguagem simples e adicionaremos novos recursos incrementalmente. A semântica e a sintaxe "inicial" da linguagem é semelhante ao da linguagem imperativa seqüencial apresentada no ínicio do capítulo. Entretanto, dois novos comandos foram adicionados: $\|$ e *wait*.

O comando $s_1 \| s_2$ representa a composição paralela dos comandos s_1 e s_2, i.e. um processo é criado para executar um dos comandos. A execução de $s_1 \| s_2$ termina precisamente quando s_1 e s_2 terminam. A ordem em que as ações ocorrem em s_1 e s_2, respectivamente, é preservada durante a execução de $s_1 \| s_2$. O comando *wait(expr)* bloqueia a execução do processo corrente

```
P1 -- L --> P1'
|-
(P1 || P2) -- L --> (P1' || P2).

P2 -- L --> P2'
|-
(P1 || P2) -- L --> (P1 || P2')

id L
|-
(completed || completed) -- L --> completed.

Expr -- L -->> true
|-
wait Expr -- L --> completed.
```

Figura 9.9: Semântica dos comandos $\|$ e *wait*

até que a expressão *exp* torne-se verdadeira. A Figura 9.9 contem a especificação em *PAN* dos comandos $\|$ e *wait*.

O programa formado apenas por *wait(false)* termina anormalmente, visto que nenhuma transição pode ser realizada. Entretanto, o leitor pode achar que seria mais razoável o programa entrar em *loop*. Para obtermos esta semântica para o comando *wait*, basta adicionarmos a seguinte regra:

```
Expr -- L -->> false
|-
wait Expr -- L --> wait Expr.
```

Considerando esta regra o programa *wait(false)* fica em *loop*. Com estas novas regras, temos um simulador para a linguagem concorrente. Entretanto, para a construção do analisador utilizamos uma semântica de baixo nível da mesma forma que no caso do analisador para a linguagem imperativa simples. Porém, no caso da linguagem concorrente, podemos ter mais de um processo ativo. Portanto, a semântica de baixo nível utiliza uma seqüência de *instruction pointers* ao invés de apenas um, como é o caso da linguagem imperativa seqüencial. Quando um processo é criado, um novo *instruction pointer* é adicionado a seqüência.

Para evitarmos a explosão do número de estados devido ao *interleaving*, o nosso analisador utiliza a técnica de *partial order methods* descrita na seção 3.5.5. A identificação de transições *não interferentes* é simples neste caso, já que a linguagem não possui recursos de *aliasing* [1] de variáveis.

Também podemos definir uma linguagem (representação) intermediária para a linguagem concorrente apresentada nesta seção. Esta linguagem intermediária é baseada na linguagem intermediária associada a linguagem imperativa simples. A diferença é que no lugar de um *instruction pointer*, o *estado da computação* pode ter agora uma seqüência de *instruction pointers*. Isto é, um *instruction pointer* para cada processo ativo.

A princípio pode parecer que a "semântica" descrita na Figura 9.5 não poderá ser reutilizada, visto que a mesma faz menção a apenas um *instruction pointer*, vide os predicados: *get-ip*, *set-ip*

[1] Alocação dinâmica de memória por exemplo.

e *next*. Entretanto, como a estrutura do *estado da computação* está encapsulada, precisamos apenas:

- Modificar a estrutura do *estado da computação* para esta conter um seqüência de *instruction pointers*
- Adicionar o "conceito" de processo "corrente" ao *estado da computação*. O processo "corrente" é aquele que terá a chance de realizar uma transição. Obviamente, a cada momento pode existir apenas um processo "corrente".
- Modificar a semântica dos predicados *get-ip* e *set-ip*. Agora, o predicado *get-ip* "pega" o *ip* corrente do processo "corrente". O predicado *set-ip* modifica o valor do *ip* corrente do processo "corrente". O predicado *next* não precisa ser modificado, uma vez que este usa os predicados *get-ip* e *set-ip*.
- Adicione o predicado (não determinístico) *select-current-process*.

Com estas modificações, podemos definir a "semântica" da representação intermediária da linguagem concorrente. Assuma que o predicado `_ -- _ --> _` define a relação de transição da linguagem intermediária definida na Figura 9.5, e o predicado `_ -- _ -->p _` define a relação de transição da linguagem intermediária concorrente. Assim temos:

```
ST -- L -->p ST' :-
        select-current-process ST STtmp1,
        STtmp1 -- L --> ST'.
```

Note que é possível definir apenas um predicado de transição, mas para isso, definimos o predicado *deselect-current-process*, que faz com que não haja nenhum processo "corrente". O predicado *get-ip* falha quando não há nenhum processo "corrente" selecionado. Com estas modificações, o predicado *select-current-process* falha se já houver um predicado "corrente" já "selecionado". Assim temos:

```
ST -- L --> ST' :-
        select-current-process ST STtmp1,
        STtmp1 -- L --> STtmp2,
        deselect-current-process STtmp2 ST'.
```

Sem estas modificações, o programa entraria em *loop*, já que *select-current-process* nunca falharia, e a regra acima poderia ficar invocando a si mesma para sempre.

Como mencionado anteriormente, utilizamos *partial order methods* para conter a explosão do número de estados. Apesar da definição desta técnica envolver a identificação de transições *não interferentes*, o nosso protótipo identifica *opcodes não interferentes*. Um *opcode* é *não interferente* quando nenhuma transição associada ao mesmo é interferente com uma transição associada a um *opcode* de outro processo. A partir desta simplificação, precisamos apenas classificar os *opcodes* e modificar a semântica do predicado *select-current-process* da seguinte forma:

- Se existe um ou mais processos cujo o *ip* corrente está associada a um *opcode não intereferente*, então escolha um destes processos não deterministicamente e realize o *cut* em seguida.
- Caso contrário, apenas selecione um processo não deterministicamente.

9.7.1 Adicionando alocação dinâmica de memória

Na linguagem paralela descrita acima é extremamente simples determinar se duas transições são ou não *interferentes*. Porém, em uma linguagem que suporta *aliasing* a situação se complica muito. Uma linguagem suporta *aliasing* quando esta permite que uma área da memória seja acessada de duas ou mais formas [68]. Por exemplo, uma variável de C pode ter seu endereço computado e o seu valor lido e modificado pelo nome ou através de um ponteiro. Portanto, é necessário informação sobre *aliasing* para determinar se duas transições são ou não interferentes. Por exemplo, considere o programa a seguir:

```
x := 0;
y := &x;
(x := 1 || *y := 2)
```

A expressão $\&x$ retorna o endereço da variável x, e o comando *y := 2*, modifica a área de memória apontada por y. Portanto, é claro que os comandos x := *1* e *y := *2* são dependentes [2], desde que ambos modificam o valor da variável x.

A informação sobre *aliasing* é em geral computada através de um *alias analyzer*. Como descrito anteriormente, nossos analisadores de programas reduzem o mapa de estados usando informação sobre transição interferentes. É claro que temos aqui um problema "recursivo", desde que para determinar se duas transições são interferentes ou não, precisamos de informação sobre *aliasing*; e para computar (efetivamente) a informação sobre *aliasing* precisamos de um analisador que utiliza *partial order methods*, e por sua vez precisa determinar se duas transições são ou não interferentes. Entretanto, é possível obter informação menos precisa sobre *aliasing* usando um analisador insensível ao fluxo de controle [68]. Como o analisador é insensível ao fluxo de controle, o *interleaving* não o afeta.

Para computar informação insensível ao fluxo de controle sobre *aliasing*, utilizamos uma técnica baseada em *set constraints* [37]. Esta abordagem baseada em *set constraints* pode ser vista como uma abstração das abordagens propostas por Andersen [5] e Steensgaard [92]. O analisador de Andersen é mais preciso e possui complexidade $O(n^3)$, onde n é o tamanho do programa. A analisador de Steensgaard é menos preciso, mas a complexidade é quase linear $O(n\alpha(n,n))$, onde α é a inversa da função de Ackerman.

Os dois tipos de analisador constroem um grafo contendo informação sobre *points-to*, i.e. que variável aponta para qual. A partir desta informação é trivial obter informação sobre *aliasing*. Os nodos do grafo representam posições de memória ou conjuntos de posição de memória. Um arco entre os nodos x e y representa a situação onde uma localização de x aponta para uma localização de y. Posições de memória podem ser variáveis locais e globais, endereços de funções e objetos do *heap*.

O nosso protótipo utiliza uma análise baseada no analisador de *Andersen*. O resultado obtido com este analisador pode, então, ser utilizado para determinar se duas transições são interferentes ou não, habilitando-nos a realizar analises mais precisas através da técnica de *partial order methods*. O algoritmo baseado em *set constraints* é detalhado no Apêndice C.

9.7.2 Alocação Dinâmica de Processos

A maioria das linguagens paralelas possuem um conjunto de primitivas tal como *fork* e *spawn*, que permitem a criação de um número ilimitado de processos. Logo, algum tipo de aproximação deve ser utilizada para garantir a terminação dos analisadores.

[2] Isto é, os comandos interferem um com o outro.

Como foi mencionado anteriormente, o estado de um programa paralelo contem uma seqüência de *instruction pointers* que indicam que parte do programa cada processo está executando. Por exemplo, o estado inicial do programa:

$$\textcircled{1}\ \mathrm{x} = 0\ \textcircled{2} \parallel \textcircled{3}\ \mathrm{y} = 0\ \textcircled{4} \parallel \textcircled{5}\ \mathrm{z} = 0\ \textcircled{6}$$

pode ser representado como "[1,3,5] (x = $\top$, y = $\top$, z = $\top$)", onde [1,3,5] é o conteúdo da seqüência de *instruction pointers*. O estado final deste programa é "[2,4,6], (x = 0, y = 0, z = 0)".

Apesar de estarmos utilizando o termo *seqüência de instruction pointers*, do ponto de vista semântico, o que temos é uma *bag* (conjunto com repetições). Portanto, a partir deste ponto usaremos o termo *bag de instruction pointers*.

A *bag de instruction pointers* pode ser representada por uma função, que dado um particular valor de *instruction poiter* retorna o número de processos nesta posição (*Instruction-Pointer* $\rightarrow$ *Nat*). Por exemplo, a *bag* [2,4,4,4,5,5] seria representada pela função f, tal que:

$$f(ip) = \begin{cases} 1 & se\ ip = 2 \\ 3 & se\ ip = 4 \\ 2 & se\ ip = 5 \\ 0 & caso\ contrário \end{cases}$$

A partir desta representação, podemos definir uma *bag abstrata de instruction pointers* como sendo uma função que dado um particular valor de *instruction poiter* retorna um elemento do conjunto $\{0, 1, \infty\}$. O símbolo ∞ representa o caso de existirem 2 ou mais processos em um determinado *instruction pointer*. Como o domínio e o codomínio desta função são finitos, temos então um número finito de *bags abstratas de instruction pointers*, garantindo, portanto, a terminação de nossos analisadores.

A função α de abstração de *bags de instruction pointers* pode ser definida como:

$$\begin{array}{l} \alpha : Bag \rightarrow Abstract\text{-}Bag \\ \alpha(bag) \quad = \quad abs\text{-}bag\ onde \\ \qquad\qquad abs\text{-}bag(ip) = \begin{cases} 0 & se\ bag(ip) = 0 \\ 1 & se\ bag(ip) = 1 \\ \infty & caso\ contrário \end{cases} \end{array}$$

Para facilitar a exposição, utilizamos a notação $[4, 10^+]$ para representar uma *bag abstrata de instruction pointers abs-bag* tal que:

$$abs\text{-}bag(ip) = \begin{cases} 1 & se\ ip = 4 \\ \infty & se\ ip = 10 \\ 0 & caso\ contrário \end{cases}$$

Usamos a notação $[\ldots, ip, \ldots]$ para representar uma *bag abstrata de instruction pointers abs-bag* onde $abs\text{-}bag(ip) = 1$. Analogamente, usamos $[\ldots, ip^+, \ldots]$ para representar o fato de que $abs\text{-}bag(ip) = \infty$. O estado abstrato da computação será representado por $\langle abs\text{-}bag, abs\text{-}store \rangle$.

Usando estas definições, podemos especificar as modificações na semântica abstrata necessárias devido a utilização de *bags abstratas de instruction pointers*. Em suma, precisamos apenas definir novas transições para lidar como *bags abstratas* tais como: $[\ldots, ip^+, \ldots]$. Neste caso, devemos considerar duas situações possíveis:

1. Há exatamente dois processos no *instruction pointer ip*.

$$\frac{\langle [\ldots, ip, \ldots], abs\text{-}store \rangle \rightarrow \langle [\ldots, ip', \ldots], abs\text{-}store' \rangle}{\langle [\ldots, ip^+, \ldots], abs\text{-}store \rangle \rightarrow \langle [\ldots, ip, ip', \ldots], abs\text{-}store' \rangle}$$

① x := 1; ②
while (x = 1) do
    ③ newprocess (④ x := 0 ⑤); ⑥
⑦

Figura 9.10: Parallel Program with Dynamic Process Creation

2. Há mais de dois processos no *instruction pointer ip*.

$$\frac{\langle [\ldots, ip, \ldots], \textit{abs-store} \rangle \rightarrow \langle [\ldots, ip', \ldots], \textit{abs-store}' \rangle}{\langle [\ldots, ip^+, \ldots], \textit{abs-store} \rangle \rightarrow \langle [\ldots, ip^+, ip', \ldots], \textit{abs-store}' \rangle}$$

A Figura 9.10 contem um programa paralelo simples que cria dinamicamente processos. O comando "*newprocess*(*stmt*)" cria um novo processo para executar *stmt* e continua a execução do processo corrente. A Figura 9.11 contem o mapa de estados abstratos do programa. A seguir, comentamos algumas das transições do mapa de estados com o intuito de facilitar o entendimento do mesmo.

- $\langle [4, 3], x = 1 \rangle \rightarrow \langle [4^+, 6], x = 1 \rangle$, o comando *newprocess* é executado pelo processo em ③, um novo processo é criado em ④, e o processo em ③ continua a execução em ⑥.
-
- $\langle [4^+, 3], x = 1 \rangle \rightarrow \langle [4^+, 5, 3], x = 0 \rangle$, o comando $x := 0$ é executado por um dos processos em ④. Esta transição assume que há mais de dois processos em ④. Por outro lado, a transição $\langle [4^+, 3], x = 1 \rangle \rightarrow \langle [4, 5, 3], x = 0 \rangle$ assume que há exatamente dois processos em ④.
- O estado terminal $\langle [5, 7], x = 0 \rangle$ representa o *trace* onde o corpo do *loop* foi executado exatamente uma vez. O estado terminal $\langle [5^+, 7], x = 0 \rangle$ representa todos os *traces* onde o *loop* foi executado mais de uma vez.

9.7.3 Analisando Programas Paralelos "Reais"

Análise global de programas "reais" é normalmente impraticável mesmo para programas seqüências determinísticos. O custo deste tipo de análise é justificável, por exemplo, em atividades de re-engenharia. No caso de linguagens determinísticas é normalmente apenas realizada análise intraprocedural, i.e. cada procedimento é analisado independentemente. Esta abordagem pode também ser utilizada em linguagens paralelas, entretanto, devemos sempre assumir que o efeito de um *opcode interferente* é indefinido, ou seja, o mais conservativo possível. Por exemplo, um *opcode* que acesse o valor de uma variável compartilhada, deve assumir que o valor da mesma é $\top$ (indefinido), visto que processos concorrentes podem modificar o valor da mesma. Observe que, se uma variável compartilhada estiver dentro de uma *seção crítica*, esta aproximação "drástica" não é necessária.

Também é possível realizar a análise por módulos, i.e. um módulo (ou grupo de módulos) é analisado de cada vez. Neste caso, não será necessário utilizar a aproximação "drástica" definida no parágrafo anterior para as variáveis compartilhadas privadas a um módulo.

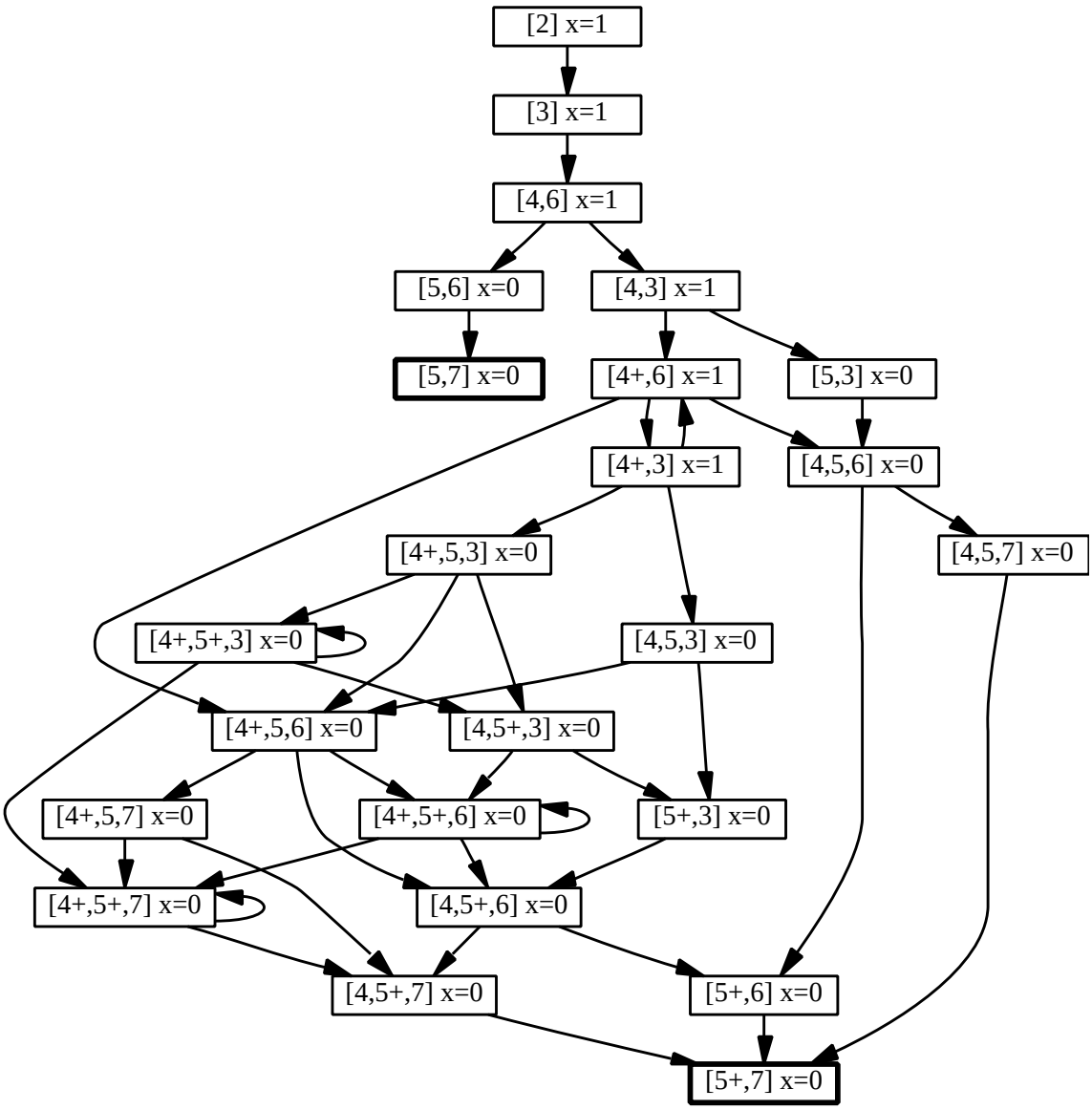

Figura 9.11: Mapa de estados do programa com alocação dinâmica de processos.

Uma análise mais precisa ainda pode ser gerada, se um modelo do ambiente de execução do módulo for utilizada. Definimos como ambiente de execução, o resto do programa e o ambiente externo ao programa. É importante lembrar, que o resultado da análise estará correto apenas se o modelo for uma representação correta (segura) do ambiente de execução.

Para exemplificar como um modelo pode melhorar os resultados da análise, considere o caso em que o modelo garante que o valor de uma variável será sempre maior que zero. Portanto, este fato pode ser utilizado ao analisarmos um *opcode* que acesse o valor desta variável.

Note que, a entrada para a ferramenta de análise é composta do módulo a ser analisado mais a especificação do ambiente de execução. Uma abordagem similar é freqüentemente utilizada em atividade de verificação [46].

9.8 Aplicando o "Framework" a uma DSL

Até agora, apresentamos vários exemplos relacionados a linguagens de propósito geral. Nesta seção, iremos aplicar o nosso "framework" a um DSL para o domínio de controle de processos industriais e controladores lógicos programáveis (PLCs) [6].

9.8.1 DSL description

PLCs podem ser programados para computar uma relação, variável no tempo, entre um conjunto de saídas controladas e um conjunto de entradas relacionadas a um processo industrial. Em 1993, o *International Electrotechnical Commission (IEC)* publicou o padrão IEC 1131 para controladores programáveis. A parte 3 do padrão [60] define uma coleção de linguagens específicas de domínio (DSLs). Entretanto, estamos apenas interessados no subconjunto relacionado aos *Sequential Function Charts* (SFCs). Estes elementos de programação foram inspirados em *Petri nets* e são utilizados para especificar a relação de entrada-saída computada por um *PLC* em resposta a eventos externos e internos.

A Figura 9.12 contem um exemplo de *SFC*. Neste exemplo, a tarefa é transferir uma certa quantidade de cascalho do granel para o caminhão mediante supervisão humana. Controle humano pode ser realizado através de um painel contendo duas chaves *on/off*, que são representadas pelas variáveis booleanas *sw* e *load*. As outras duas entradas vem de sensores representados pelas variáveis *binEmpty* e *truckOnRamp*. Baseado nestas entradas, as saídas booleanas do controlador são *run* and *binValve*, utilizadas para controlar a esteira e a válvula no fundo do granel.

SFCs são usualmente representados graficamente conforme a Figura 9.12. Existem dois tipos de nodos: retângulos denominados de "passos" (*steps*) e linhas grossas denominadas de transições. Cada "passo" tem uma ação associada, que é representada em um retângulo adjacente. Transições são rotuladas por expressões booleanas que indicam quando estas estão habilitadas. Alguns "passos" podem ser marcados como "passos iniciais". "Passos iniciais" são representados com uma borda dupla.

O programa da Figura 9.12 é composto de dois *SFCs* paralelos. O *SFC* da esquerda controla as entradas do painel de controle e o movimento da esteira. O *SFC* da direita controla o granel.

Para definirmos a semântica dos diagramas *SFC*, definimos uma representação gráfica para estes. A representação gráfica pode ser automaticamente extraída da nossa representação textual. A sintaxe da nossa representação textual é definida como:

$$P ::= \langle a \rangle \mid P \ ; \ G \mid P_1 \parallel P_2 \mid \textbf{loop}\ P : b$$

$$G ::= b \rightarrow P \mid G_1 [\!] G_2$$

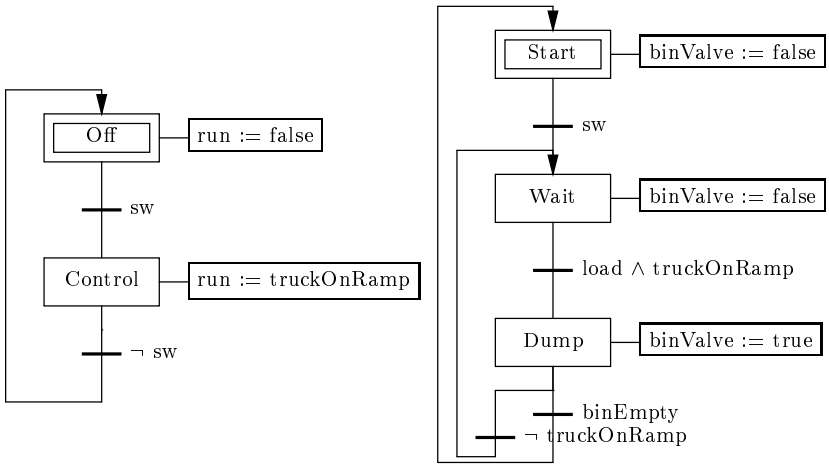

Figura 9.12: Gravel SFC

P representa os programa e G os "guardas". O termo $\langle a \rangle$ representa um "passo" associado a uma ação a, b representa uma condição para realizar uma transição. P ; G é a composição seqüencial via a transição em G. $P_1 \parallel P_2$ é a composição paralela. **loop** $P : b$ representa um *loop* com corpo P e transição b ligando ao início do *loop*. $G_1 [\!] G_2$ é a escolha não determinística entre G_1 e G_2. O diagrama da Figura 9.12 é representado como:

$$
\begin{array}{l}
\mathbf{loop}(\langle run := false \rangle \ ; \ sw \rightarrow \langle run := truckOnRamp \rangle) : \neg sw \\
\parallel \\
\mathbf{loop}(\langle binValve := false \rangle \ ; \\
\quad sw \rightarrow \mathbf{loop}(\langle binValve := false \rangle \ ; \\
\qquad\qquad (load \wedge truckOnRamp \rightarrow \langle binValve := true \rangle) \\
\qquad\quad) : \neg truckOnRamp \\
\quad) : binEmpty
\end{array}
$$

O padrão IEC1131 permite que diferentes sublinguagens sejam utilizadas para especificar as ações e expressões (condições). Entre estas linguagens temos *ladder logic* [60] e uma linguagem imperativa. Em nosso protótipo reutilizamos a linguagem imperativa descrita no início deste capítulo.

Note que estamos assumindo que o ambiente de operação de um *PLC* produz sinais em uma freqüências muito menores que a freqüência do *clock* do *PLC*. Assim, podemos assumir que a interação com o ambiente é síncrona.

9.8.2 Semântica

A Figura 9.13 contem a especificação em PAN da sintaxe abstrata da linguagem textual associada aos *SFCs*

A Figura 9.14 contem a semântica dos programas *SFC*. Observe que esta semântica depende da definição do predicado `_ -- _ -->> _` que é o fecho transitivo do predicado que define a semântica das ações e expressões. É importante observar que a semântica dos programas *SFC* está completamente desacoplada da estrutura dos *labels*! Portanto, tal descrição semântica pode ser reutilizada mesmo se linguagens diferentes, tal como *ladder logic*, para especificar as ações e expressões.

Este exemplo mostra como é possível combinar linguagens de domínio diferentes de uma forma intuitiva, desde que a estrutura dos *labels* esteja encapsulada.

```
type < _ > : action -> sfc.
type _ ; _ : sfc -> sfc -> sfc {prec 80 l-assoc}.
type _ || _ : sfc -> sfc -> sfc {prec 75 l-assoc}.
type loop _  _ : sfc -> expression -> sfc {prec 65}.
type _ -> _ : expression -> sfc -> sfc {prec 60}.
type _ [] _ : sfc -> sfc -> sfc {prec 70 l-assoc}.
```

Figura 9.13: SFC abstract syntax described in PAN

```
A -- L -->> completed
|-
< A > -- L --> completed.

P -- L --> P'
|-
(P ; G) -- L --> (P' ; G).

id L |- (completed ; P ) -- L --> P.

P1 -- L --> P1'
|-
(P1 || P2) -- L --> (P1' || P2).

P2 -- L --> P2'
|-
(P1 || P2) -- L --> (P1 || P2')

id L |- (completed || completed) -- L --> completed.

id L |- (loop P B) -- L --> (P ; B -> (loop P B)).

B -- L -->> true
|-
(B -> P) -- L --> P.

G1 -- L --> G1'
|-
(G1 [] G2) -- L --> G1'.

G2 -- L --> G2'
|-
(G1 [] G2) -- L --> G2'.
```

Figura 9.14: SFC semantics

9.8.3 Gerando ferramentas

SFC é um linguagem simples e a maior parte das aproximações estão associada com a linguagem utilizada para descrever ações e expressões. A única aproximação que devemos nos preocupar é a relacionada com expressões, uma vez que o valor das expressões pode não ser preciso, i.e. um valor *booleano* aproximado pode ser igual a $\top$ (top). Portanto, adicionamos uma transição extra refletindo este caso. Esta transição é definida como:

```
B -- L -->> top
|-
(B -> P) -- L --> P.
```

Simulações e verificações também podem ser realizadas. Entretanto, se o usuário está interessado em realizar verificações, ele/ela também deve modelar o ambiente de execução, i.e. o sistema reativo aberto deve ser fechado. Note que o usuário pode utilizar linguagens diferentes para descrever o ambiente de execução. Este é um recurso interessante, uma vez que linguagens mais expressivas podem ser utilizadas para descrever o ambiente de execução.

É importante lembrar que no caso da linguagem *SFC*, nenhuma linguagem intermediária foi utilizada em nosso protótipo.

9.9 Desenvolvendo *DSLs*

Neste e nos capítulos anteriores mostramos como construir analisadores e intrepretadores de forma modular com a linguagem *PAN*. Na Seção 5.3 apresentamos uma justificativa para o uso de linguagens lógicas na transformação de programas [102, 90]. Praticamente todas as etapas do desenvolvimento de uma *DSL* podem ser codificados na linguagem *PAN*. Em suma, apenas o analisador sintático da linguagem deve ser implementado por um módulo a parte. Para linguagens simples podemos utilizar a própria linguagem *PAN* para especificar o analisador sintático, utilizando uma abordagem semelhante a descrita em [90].

É importante salientar que nada impede a integração da linguagem *PAN* com outros sistemas de transformação tais como *Draco* [69, 58] e *DMS* [9]. Esta integração pode ser realizada via os mecanismos de interfaceamento com a linguagem *C* existentes na linguagem *PAN*.

9.10 Conclusão

Neste capítulo apresentamos diversos exemplos de simuladores, analisadores e verificadores de programas que foram implementados dentro de nosso "framework". Mostramos também como representações intermediárias podem ser utilizadas para aumentar a eficiência das ferramentas geradas. Em especial, também apresentamos técnicas para lidar com linguagens contendo alocação dinâmica de memória e linguagens concorrentes.

9.10.1 Contribuições

- Uma abordagem para a implementação de analisadores para linguagens concorrentes. É importante lembrar que existem poucos trabalhos sobre analisadores para linguagens concorrentes [99, 81, 104]. E estes não abordam linguagens concorrentes contendo alocação dinâmica de memória ou processos.

- Uso de técnicas de *set constraints* em conjunto com técnicas de Interpretação Abstrata e Verificação de Modelos para a análise programas concorrentes.

- Uso de modelos do ambiente de execução para melhorar o resultado da análise.
- Uso de marcações para definir aproximações e contextos de verificação.
- Aplicação incomum de técnicas de Verificação de Modelos para a detecção de erros tais como: dereferenciação de ponteiros nulos, índices inválidos de *arrays*, etc.

Capítulo 10

Trabalhos Correlatos

10.1 Introdução

O nosso "framework" suporta a construção de analisadores e verificadores de código a partir de especificações de elevado nível da abstração. Por esta razão, a tarefa de comparar o nosso sistema com outros trabalhos não é simples. Decidimos comparar a nossa abordagem com "frameworks" para a construção de analisadores baseados em semântica, ferramentas de verificação específicas e outras abordagens para análise de código. Contudo, não conseguimos encontrar na literatura nenhuma ferramenta de verificação que seja customizável pela semântica da linguagem.

10.2 *Data Flow Analyzers* baseados em semântica denotacional

Na literatura existem diversas propostas de "frameworks" [17, 52, 73, 98, 34, 54] baseados em semântica denotacional, para a geração de analisadores de programas. Entretanto, como já foi dito, dentro do nosso ponto de vista não faz muito sentido basear procedimentos de análise inerentemente intencionais (envolvendo noções como passo computacional, transição, ...) como Interpretação Abstrata e Verificação de Modelos, numa teoria extensional como semântica denotacional.

Acreditamos que a motivação para estes trabalhos esta baseada em um entendimento distorcido do que seja semântica denotacional. Em semântica denotacional, a semântica de um programa é dada através da apresentação de uma estrutura matemática que capture o significado do mesmo. Em geral, as especificações denotacionais utilizam *CPOs* (*Complete Partial Orders*) e funções contínuas [85], embora outras estruturas matemáticas também possam ser utilizadas, tais como *Metric Spaces* [31] ou *Pomsets* [80]. Para facilitar a construção de especificações denotacionais, i.e. mapeamentos entre programas e denotações, meta-linguagens são utilizadas. A meta-linguagem funciona como uma "ponte" entre a linguagem de programação e a estrutura matemática (Figura 10.1). É óbvio que, este "truque" só é útil se o mapeamento da linguagem de programação para a meta-linguagem for mais simples que o mapeando direto para uma estrutura matemática. É importante ressaltar, que uma meta-linguagem *não* precisa necessariamente possuir uma interpretação operacional.

Em geral, a meta-linguagem utilizada em descrições denotacionais é uma variação de λ-

Figura 10.1: Semântica denotacional

calculus (ex.: λ-calculus tipado + operador de ponto fixo). Como estas variações de λ-calculus possuem, em geral, uma interpretação operacional, é muito comum encontrar na literatura comentários tais como:

> *"It is well known that a denotational specification of a programming language gives, for free, an executable interpreter for this language" [54]*

Este tipo de comentário apenas faz sentido, se na especificação denotacional foi utilizada uma meta-linguagem como "ponte" semântica, e se essa meta-linguagem possui uma interpretação operacional. O nosso ponto de vista é que a maioria, senão todos, os "frameworks" baseados em semântica denotacional estão, na verdade, baseados na interpretação operacional da meta-linguagem utilizada (i.e. λ-calculus). Ou seja, a maior parte destes trabalhos pode ser visto como um "framework" baseado em *semântica operacional* que utiliza uma variação de λ-calculus como meta-linguagem.

A utilização de especificações denotacionais também traz outros problemas. Em linguagens imperativas simples, por exemplo, a noção de passo computacional é abstraída, i.e. a semântica de um programa é dada por uma função que leva dados de entrada a dados de saída, não existindo a noção de passo computacional. Como já foi dito na Seção 4.8, esta abstração dos passos computacionais permite uma forma extensional de equivalência, como por exemplo:

$$D'[\![x := 1; y := y + 1;]\!] \neq D'[\![x := 1; y := 2;]\!]$$

Contudo, ao construirmos analisadores, estamos interessados nestes "detalhes" (i.e. nos passos computacionais) que foram "abstraídos". Para resolver estes problemas, todas as abordagens baseadas em semântica denotacional exigem que a semântica da linguagem seja modificada com o intuito de coletar informação sobre os passos computacionais.

Como estes "frameworks" estão na verdade utilizando meta-linguagens com interpretação operacional. Todos eles podem ser mapeados para a nossa abordagem, da mesma forma que semântica de ações foi mapeada (Seção E). Em suma, precisamos apenas especificar, em *PAN*, a semântica operacional da meta-linguagem utilizada. Obviamente, todos os problemas de eficiência relacionados ao uso de meta-linguagens genéricas (Seção 5.6.2.1) se aplicam a este caso.

10.2.1 "Framework" denotacional de *Nielson*

O "framework" denotacional de *Nielson* [73] é puramente conceital, não possuindo implementação. O sistema de *Gouge* [34] e a ferramenta *SPARE* [98] são implementações deste "framework".

Neste "framework" o usuário precisa adaptar uma especificação denotacional com o inuito de coletar informações sobre os passos computacionais. Considere a especificação denotacional contida nas Figuras 10.2, 10.3 e 10.4 de uma linguagem imperativa simples.

A informação associada aos passos computacionais é armazenada nos pontos de controle do programa, definindo um tipo de *collecting semantics*. Um ponto do programa pode ser especificado por uma tupla $\langle occ, q\rangle \in Pla$. Os elementos occ são utilizados para rotular nodos da árvore de sintaxe abstrata. A raiz é rotulada como $\langle\ \rangle$ e o i-ésimo filho de um nodo rotulado como occ é rotulado $occ\S\langle i\rangle$. O elemento q é útil para especificar quando um ponto de controle está a direita ou a esquerda de um nodo.

Com o uso deste mapeamento dos nodos da árvore de sintaxe abstrata para os elementos occ, poderemos, então, modificar as equações semânticas para que elas associem informação aos pontos de controle do programa (os elementos $\langle occ, q\rangle$). Entretanto, as funções semânticas $\mathcal{C}$ e $\mathcal{E}$ precisam de um novo parâmetro especificando o elemento occ em questão. A função *attach*

$$
\begin{array}{l}
\mathcal{C} \in Cmd \to Cont \to Cont \\
\mathcal{C}[\![cmd_1; cmd_2]\!]\ c = \\
\quad \mathcal{C}[\![cmd_1]\!]\ (\mathcal{C}[\![cmd_2]\!]\ c) \\
\mathcal{C}[\![ide := exp]\!]c = \\
\quad \mathcal{E}[\![exp]\!]\ (assign\ [\![ide]\!]\ c) \\
\mathcal{C}[\![\textbf{if}\ exp\ \textbf{then}\ cmd_1\ \textbf{else}\ cmd_2\ \textbf{fi}]\!]\ c = \\
\quad \mathcal{E}[\![exp]\!]\ (cond\ (\mathcal{C}[\![cmd_1]\!]\ c)\ (\mathcal{C}[\![cmd_2]\!]\ c)) \\
\mathcal{C}[\![\textbf{while}\ exp\ \textbf{do}\ cmd\ \textbf{od}]\!]\ c = \\
\quad FIX(\lambda c'.\mathcal{E}[\![exp]\!]\ (cond\ (\mathcal{C}[\![cmd]\!]\ c')c)) \\
\mathcal{C}[\![\textbf{write}\ exp]\!]\ c = \\
\quad \mathcal{E}[\![exp]\!]\ (write\ c) \\
\mathcal{C}[\![\textbf{read}\ ide]\!]\ c = \\
\quad read(assign[\![ide]\!]\ c) \\
\mathcal{E} \in Exp \to Cont \to Cont \\
\mathcal{E}[\![\mathrm{exp}_1\ \ ope\ \ \mathrm{exp}_2]\!]\ c = \\
\quad \mathcal{E}[\![\mathrm{exp}_1]\!](\mathcal{E}[\![\mathrm{exp}_2]\!](apply\ [\![ope]\!]\ c)) \\
\mathcal{E}[\![ide]\!]\ c = \\
\quad content\ [\![ide]\!]\ c \\
\mathcal{E}[\![bas]\!]\ c = \\
\quad push[\![bas]\!]\ c
\end{array}
$$

Figura 10.2: Equações Semânticas

Val	$=$	$T + N + \ldots + \{\text{“}nil\text{”}\}$	variables
Env	$=$	$Ide \to Val$	environments
Inp	$=$	Val^*	inputs
Out	$=$	Val^*	outputs
Tem	$=$	Val^*	temporary result stack
Sta	$=$	$Env \times Inp \times Out \times Tem$	states
Ans	$=$	$Out + \{\text{“}error\text{”}\}$	answers
$Cont$	$=$	$Sta \to Ans$	continuations
Occ	$=$	N^*	occurrences
Pla	$=$	$Occ \times Q$	places

Figura 10.3: Domínios Semânticos

$$
\begin{array}{l}
apply\ [\![ope]\!] \in Cont \to Cont \\
assign\ [\![ide]\!] \in Cont \to Cont \\
content\ [\![ide]\!] \in Cont \to Cont \\
push\ [\![bas]\!] \in Cont \to Cont \\
write\ [\![exp]\!] \in Cont \to Cont \\
read\ [\![ide]\!] \in Cont \to Cont \\
assign\ [\![ide]\!] \in Cont \to Cont \\
cond \in Cont \times Cont \to Cont
\end{array}
$$

Figura 10.4: Funções Auxiliares

é utilizada para associar informação a um ponto de controle específico. As Figuras 10.5 e 10.6 contêm parte das moficações necessárias. A escolha do domínio *Ans* descrito na Figura 10.6 não é única, por exemplo em [34] o domínio *Ans* é definido como $Ans = (Pla \times Sta)^*$.

Esta especificação semântica modificada [1] coleta informações de uma particular execução do programa. Alguns domínios semânticos e funções auxiliares são substituídos para capturar informação de diversas execuções, i.e. a especificação passa a lidar com conjuntos de estados. A Figura 10.7 contém algumas das modificações necessárias.

O último passo para a definição do analisador é substituir os domínios concretos por abstrações. Podemos substituir, por exemplo, o domínio $\wp(Sta)$ pelo domínio *StaSign*, onde o valor das variáveis é substituído pelo seu sinal.

10.2.2 Comparação com o nosso sistema

Como já mencionado, essas abordagens "denotacionais" são, na verdade, baseadas numa interpretação operacional. Esta visão é inclusive compartilhada pelos desenvolvedores da ferramenta *SPARE* [98]:

> *"To reason about the information obtained from an analysis specification, we need to provide an operational definition of the specification language"*
>
> *"The formal semantics for the SPARE specification language are specified using a variant of Natural Semantics ..."*

Portanto, as meta-linguagens utilizadas por estas abordagens "denotacionais" podem ser mapeadas para a nossa abordagem. Indepentende disto, identificamos também as seguintes deficiências nestas abordagens e/ou ferramentas:

- A convergência do analisador é garantida apenas para especificações denotacionais onde a aplicação do operador de ponto fixo equivale a uma recursão de cauda. Não é possível, assim, realizar análises interprocedurais com esta abordagem.
- Nenhum dos trabalhos que foram referenciados possui suporte a aplicação de operadores de aceleração de convergência (*widening* e *narrowing*). Porém, estes operadores podem ser adaptados para estas abordagens.
- A análise de linguagens concorrentes não é abordada. Um dos motivos é que a semântica denotacional de linguagens concorrentes é mais complexa, i.e. é necessário o uso de *resumptions*.

[1] Esta semântica também é denominada de *collecting semantics*.

$$
\begin{array}{l}
\mathcal{C} \in \mathit{Cmd} \to \mathit{Occ} \to \mathit{Cont} \to \mathit{Cont} \\
\mathcal{C}[\![\textbf{if } \mathit{exp} \textbf{ then } \mathit{cmd}_1 \textbf{ else } \mathit{cmd}_2 \textbf{ fi}]\!]\ \mathit{occ}\ c = \\
\quad \mathit{attach}\ \langle \mathit{occ}, \text{“}L\text{”} \rangle\ (\\
\quad \mathcal{E}[\![\mathit{exp}]\!]\ (\mathit{occ}\S\langle 1\rangle)\ (\\
\quad\ \mathit{cond}\ (\mathcal{C}[\![\mathit{cmd}_1]\!]\ (\mathit{occ}\S\langle 2\rangle)\ (\mathit{attach}\ \langle \mathit{occ}, \text{“}R\text{”} \rangle\ c)) \\
\quad\quad (\mathcal{C}[\![\mathit{cmd}_2]\!]\ (\mathit{occ}\S\langle 3\rangle)\ (\mathit{attach}\ \langle \mathit{occ}, \text{“}R\text{”} \rangle\ c)))) \\
\mathcal{C}[\![\textbf{while } \mathit{exp} \textbf{ do } \mathit{cmd} \textbf{ od}]\!]\ \mathit{occ}\ c = \\
\quad \mathit{attach}\ \langle \mathit{occ}, \text{“}L\text{”} \rangle\ (\\
\quad \mathit{FIX}(\lambda c'.\mathcal{E}[\![\mathit{exp}]\!]\ (\mathit{occ}\S\langle 1\rangle)\ (\\
\quad\ \mathit{cond}\ (\mathcal{C}[\![\mathit{cmd}]\!]\ (\mathit{occ}\S\langle 2\rangle)\ c') \\
\quad\quad (\mathit{attach}\ \langle \mathit{occ}, \text{“}R\text{”} \rangle\ c)))) \\
\mathcal{E} \in \mathit{Exp} \to \mathit{Occ} \to \mathit{Cont} \to \mathit{Cont} \\
\mathcal{E}[\![\mathrm{exp}_1\ \ \mathit{ope}\ \ \mathrm{exp}_2]\!]\ \mathit{occ}\ c = \\
\quad \mathit{attach}\ \langle \mathit{occ}, \text{“}L\text{”} \rangle\ (\\
\quad \mathcal{E}[\![\mathrm{exp}_1]\!]\ (\mathit{occ}\S\langle 1\rangle)\ (\\
\quad\ \mathcal{E}[\![\mathrm{exp}_2]\!]\ (\mathit{occ}\S\langle 3\rangle)\ (\\
\quad\quad \mathit{apply}\ [\![\mathit{ope}]\!]\ (\\
\quad\quad\ (\mathit{attach}\ \langle \mathit{occ}, \text{“}R\text{”} \rangle\ c)))))
\end{array}
$$

Figura 10.5: Equações Semânticas Modificadas

$$
\begin{array}{lcl}
\mathit{Ans} & = & \mathit{Pla} \to \wp(\mathit{Sta}) \\
\mathit{Cont} & = & \mathit{Sta} \to \mathit{Ans}
\end{array}
$$

Figura 10.6: Domínios Semânticos Modificados

$$
\begin{array}{lcl}
\mathit{StaSet} & = & \wp(\mathit{Sta}) \\
\mathit{Ans} & = & \mathit{Pla} \to \wp(\mathit{Sta}) \\
\mathit{Cont} & = & \mathit{StaSet} \to \mathit{Ans} \\
\mathit{cond}\ c_1\ c_2 = & & \\
\quad & & \lambda s.c_1\{\mathit{sta} |\ \mathit{sta} \in s \wedge \text{“}\mathit{cond\ is\ valid\ in\ sta}''\} \\
\quad & & \quad \sqcup\ c_2\{\mathit{sta} |\ \mathit{sta} \in s \wedge \text{“}\mathit{cond\ is\ not\ valid\ in\ sta}''\}
\end{array}
$$

Figura 10.7: Domínios Semânticos e Funções Modificadas

$$\begin{cases} x_1 &= \Phi(x_1, \ldots, x_n) \\ \ldots & \\ x_n &= \Phi(x_1, \ldots, x_n) \end{cases}$$

Figura 10.8: Conjunto de equações recursivas

- Os analisadores contidos nestes trabalhos possuem imprecisões geradas por caminhos impossíveis de serem executados (vide comentário na Seção 2.3).

10.3 Syntox

O sistema *Syntox* [11] é um *debugger* abstrato para um subconjunto da linguagem *Pascal* [2]. Este sistema permite a detecção de erros relativos a intervalos de variáveis inteiras (ex.: índices de *arrays*). O sistema aproxima valores de variáveis inteiras utilizando um reticulado de intervalos semelhante ao descrito no Capítulo 2. Em [12] foi demonstrado que o sistema suporta outros tipos de aproximação, e que a implementação pode ser facilmente modificada para suportar estes novos tipos de aproximação.

O sistema permite que o usuário adicione assertivas dentro do código do programa que será depurado. Existem dois tipos de assertivas. *Invariant assertions* são condições que devem ser *sempre* satisfeitas em um determinado ponto do programa. *Intermittent assertions* são condições que eventualmente devem ser satisfeitas em um determinado ponto do programa. Por exemplo, a *invariant assertion* "*false*" pode ser utilizada para especificar que um determinado ponto do programa nunca é alcançado e a *intermittent assertion* "*true*" pode ser utilizada para especificar que um determinado ponto do programa é eventualmente alcançado.

Para realizar a depuração abstrata do programa, o sistema *Syntox* converte o program em um conjunto recursivo de equações (Figura 10.8). O sistema utiliza um algoritmo de cálculo do menor ponto fixo mais sofisticado do que o utilizado no exemplo da Figura 2.12. O algoritmo ingênuo utilizado para calcular este exemplo, consiste em aplicar cada equação em paralelo até que o vetor de valores abstratos estabilize, iniciando do menor valor ($\bot$) do reticulado. Entretanto, este algoritmo ingênuo não segue o fluxo de controle do programa e recomputa as propriedades de todos os pontos de controle e cada iteração. O algoritmo do sistema *Syntox* evita este cálculos desnecessários, calculando as dependências entre as equações [13]. Esta técnica pode ser vista como uma variação do algoritmo apresentado em [68].

Para garantir a convergência do processo, o sistema utilizado operadores de *widening*, transformando, então, as equações da Figura 10.8 em $x_i = x_i \bigtriangledown \Phi(x_1, \ldots, x_n)$. O sistema tenta minimizar o número de equações (*widening points*) onde o operador de *widening* é utilizado, porque o uso excessivo deste operador compromete a precisão do resultado. Em [13] é apresentado um algortimo para calcular um subconjunto seguro de equações, onde aplicando o operador de *widening* a convergência é garantida. O sistema também utiliza operadores de *narrowing* para melhorar a precisão da análise. Para exemplificar o funcionamento do sistema *Syntox*, considere o programa exemplo descrito na Figura 10.9. Este programa é mapeado no conjunto de equações recursivas descrito na Figura 10.10. Para garantir a convergência da análise, o sistema utiliza um operador de *widening* na equação x_2, transformando a equação em $x_2 = x_2 \bigtriangledown ([\![i \leq 100]\!](x_1) \sqcup [\![i \leq 100]\!](x_3))$.

[2] Programas não contendo o operador @ (i.e. *address of*), e que não possuam procedimentos como parâmetros de outros procedimentos.

```
program Tst
    var i : integer;
begin
⓪  i := 0; ①
    while (i ≤ 100) do
    ②      i := i + 1; ③
④
end.
```

Figura 10.9: Programa exemplo para o *Syntox*

$$
\begin{aligned}
x_0 &= \top \\
x_1 &= [\![i := 0]\!](x_0) \\
x_2 &= [\![i \leq 100]\!](x_1) \sqcup [\![i \leq 100]\!](x_3) \\
x_3 &= [\![i := i + 1]\!](x_2) \\
x_4 &= [\![i > 100]\!](x_1) \sqcup [\![i > 100]\!](x_3)
\end{aligned}
$$

Figura 10.10: Equações recursivas associadas ao programa exemplo

Durante o cálculo interativo do ponto fixo, x_2 assumirá os seguintes valores:

$$\bot, \quad [0,0], \quad [0,0] \bigtriangledown ([0,0] \sqcup [1,1]) = [0,\infty]$$

Em seguida na fase de *narrowing* temos:

$$[0,\infty], \quad [0,\infty] \bigtriangleup ([0,0] \sqcup [0,100]) = [0,100]$$

Produzindo o resultado ótimo [0,100].

Os resultados gerados pelo sistema *Syntox* podem ser utilizados, também, para otimizar código. Por exemplo, compiladores *Pascal*, em geral, geram código, tal que, para cada acesso a um elemento de um *array* é verificado se o índice de acesso é um índice válido. Os intervalos computados pelo sistema *Syntox* podem ser utilizados para remover com segurança estas verificações de índice de *arrays*.

Para realizar análise interprocedural de procedimentos recursivos, o sistema *Syntox* expande o conjunto de equações dinamicamente durante o processo de análise. O conjunto de equações é expandido em dois casos:

- na primeira vez que um procedimento p é invocado;
- quando um procedimento p é invocado novamente, mas o *shape* da pilha abstrata é diferente das invocações anteriores a este procedimento. O termo "*shape*" é definido como a partição dos identificadores acessíveis por um procedimento em conjuntos de identificadores compartilhando a mesma localização na pilha.

10.3.1 Comparação com o nosso sistema

Se o objetivo é detectar *erros* e depurar abstratamente o programa, acreditamos que a abordagem utilizada pelo sistema *Syntox* é inferior a nossa pelos seguintes motivos:

- Em nosso sistema é possível verificar propriedades muito mais complexas, i.e., propriedades temporais (Seção 5.6.2.4). O sistema *Syntox* não possui nenhum marcador semelhante ao nosso *assure* (Seção 5.6.2.5). Outro detalhe importante é que nenhum comentário foi encontrado na documentação do sistema *Syntox* sobre problemas relacionados as *intermittent assertions*. Como já comentamos na Seção 5.6.2.3 o uso de aproximações mascara a não validade deste tipo de assertiva. Ao usarmos aproximações, estamos adicionando comportamentos adicionais, logo, uma propriedade do tipo "eventualmente c é válida no ponto x" que não é satisfeita na interpretação concreta, pode passar a ser satisfeita pela interpretação abstrata.

- As equações geradas pelo sistema *Syntox* são baseadas no diagrama de fluxo de controle, logo, elas estão sujeitas a imprecisões causadas por caminhos impossíveis de serem executados (vide comentário na Seção 2.3).

- A nossa abordagem se aplica a linguagens concorrentes. Dificilmente um método baseado em equações recursivas, pode ser adaptado para linguagens concorrentes.

- Para garantir convergência o sistema *Syntox* utiliza aproximações imprecisas. Como já foi dito, para detectarmos erros um sistema não precisa gerar informação que reflita todos os possíveis *traces* de um programa. No nosso sistema podemos utilizar aproximações mais precisas, ou até mesmo não utilizá-las. Podemos utilizar os algoritmos de detecção de divergência ou até mesmo ignorar a questão da convergência.

- O sistema *Syntox* pode analisar apenas um subconjunto de *Pascal*.

Por outro lado, se o objetivo é coletar informações para otimizar código, as aproximações imprecisas utilizadas pelo sistema *Syntox* são satisfatórias, já que neste caso é fundamental garantir a convergência do processo, e que o resultado obtido pela análise reflita todos os possíveis *traces* do programa. Nesta situação, podemos fazer a seguinte equivalência entre as técnicas utilizadas pelo *Syntox* e as providas pelo nosso sistemas:

- As equações são o equivalente ao nosso sistema de transição.

- A minimização do uso do operador de *widening* no sistema *Syntox* pode ser comparada ao nosso algoritmo, que minimiza o uso do operador *widening* (Figura 2.19).

- O processo de geração dinâmica de equações pode ser comparado ao algoritmo descrito na Seção 5.6.3.1, aonde o *shape* da pilha nada mais é do que o contexto de uso de um procedimento.

Apesar desta analogia, vale lembrar que em nosso sistema é possível obter resultados mais precisos, que não são afetados por caminhos impossíveis. Porém, esta precisão adicional gera um impacto na eficiência do analisador.

Também é importante ressaltar que a nossa abordagem baseada em contextos de uso é mais flexível e permite um maior controle sobre a questão "precisão $\times$ eficiência".

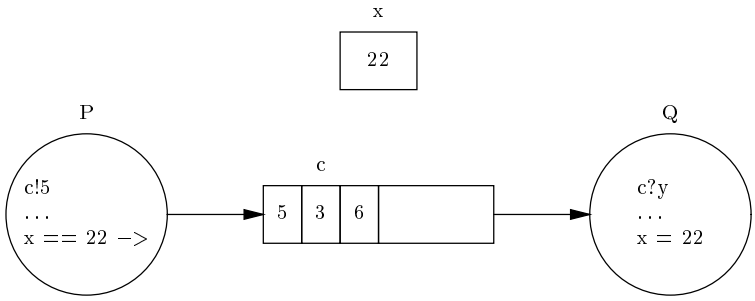

Figura 10.11: Exemplo de programa *Promela*

10.4 *Spin*

Spin [46, 47] é uma ferramenta para a verificação de sistemas concorrentes que possuam mapas de estado finitos. Os sistemas concorrentes são formalizados utilizando a linguagem *Promela*. As propriedades a serem verificadas são especificadas utilizando marcações como *assert*, e fórmulas em *LTL* (i.e. *Linear Temporal Logic*. A ferramenta *Spin* também possui um simulador (interpretador) que pode ser utilizado para executar as especificações passo a passo. Este recurso é particularmente interessante para executar *traces* onde foram detectados erros.

Ainda que *Promela* seja uma linguagem de programação, esta pode ser utilizada para formular sistemas concorrentes envolvendo *software*, *hardware* e objetos físicos. Como exemplo de objeto que pode ser especificado em *Promela*, podemos citar o mundo físico [3] ao redor de uma nave espacial [41].

Um programa *Promela* consiste em uma coleção de processos que se comunicam via canais e variáveis compartilhadas. A Figura 10.11 contém um exemplo de programa *Promela* contendo dois processos `P` e `Q`, que se comunicam via o canal `c` e a variável compartilhada `x`.

O processo `P` envia valores para o canal `c` (`c!5`) e o processo `Q` consome os valores de `c`, e os coloca na variável `y` (`c?y`). O processo `Q` atribui um valor a variável `x` ("=" representa a atribuição em *Promela*) e o processo `P` le o valor da variável `x` ("==" representa o teste de igualdade). Este exemplo ilustra os três principais componentes da linguagem *Promela*, que são variáveis, processos e canais.

Promela possui três tipos básicos: inteiro (32 bits), byte (8 bits) e booleano (1 bit). Além disso, a linguagem também permite a definição de novas estruturas de dados. Entretanto, *Promela* não possui ponteiros e nem suporte a alocação dinâmica de memória.

Os canais são *buffers* "*first in first out*" (*FIFO*) cuja capacidade máxima de mensagens deve ser especificada. Processos se comunicam escrevendo e lendo mensagens nestes *buffers*. A seguinte declaração introduz um canal denominado `c` capaz de armazenar 6 mensagens do tipo `int`.

```
chan c = [6] of {int};
```

As operações `empty`, `nempty` e `len` podem ser utilizadas para examinar o estado de um canal.

A linguagem permite a criação dinâmica de processos, entretanto, o sistema possui um número máximo (255) de processos que podem ser criados. A função deste limite é garantir que o mapa de estados possua tamanho finito.

Por se tratar de uma linguagem para especificação de modelos, *Promela* possui diversas primitivas para modelar o não determinismo. Por exemplo, o *statment if* seleciona não deterministicamente uma das alternativas cuja a condição é satisfeita (vide exemplo a seguir).

[3] Uma abstração do mundo físico que é relevante para o comportamento da nave espacial.

```
if
:: x > 0 -> ...
:: x > 10 -> ...
:: x < 0 -> ...
:: x < -10 -> ...
fi
```

Apesar de possuir suporte a processos, a linguagem *Promela* não possui o conceito de procedimentos (ou funções), o que dificulta a modelagem de diversos sistemas. O propósito desta limitação é garantir que o mapa de estados do sistema seja finito.

A ferramenta *Spin* possui as seguintes técnicas de análise:

- Abordagem ingênua: o mapa de estados é construído explicitamente.
- Execução randômica: o mapa de estados é visitado randômicamente. A cobertura de todo mapa de estados não é garantida.
- *Supertrace*: possibilita a análise de especificações mais complexas, mas sem garantir cobertura total (Seção 3.5.2).
- *Partial order methods* (Seção 3.5.5).

10.4.1 Comparação com o nosso sistema

Embora *Spin* seja uma das ferramentas mais utilizadas na prática, este possui diversas deficiências. A mais grave é permitir apenas a verificação de sistemas com mapas de estados finitos. Além disso, a linguagem *Promela* não possui suporte a funções e nem a alocação dinâmica de memória. Por outro lado, os recursos de abstração [4] existentes em nossa abordagem permitem que sistemas extremamente complexos sejam verificados. Além de tudo, permitimos a descrição de linguagens que possuam mapas de estados infinitos. Através da utilização das nossas técnicas de detecção de divergência, podemos garantir a terminação da verificação.

Uma ferramenta como a proposta nesta tese é de grande utilidade para diversos projetos [41], onde sistemas codificados em linguagens específicas tem que ser mapeados para a linguagem *Promela*. Este mapeamento é extremamente problemático, porque a linguagem *Promela* é pouco expressiva (ex.: não contém funções). Ainda mais, o usuário é responsável por aproximar a semântica do sistema manualmente, garantindo que o sistema tenha um mapa de estados finito. Neste processo de aproximação manual, vemos os seguintes problemas:

- O usuário pode cometer erros no processo de mapeamento manual.
- Se o usuário decidir modificar as aproximações, terá que modificar o mapeamento manualmente.
- Surge um problema de consistência, pois modificações no sistema tem que ser refletidas no modelo (i.e. no código *Promela*).

Em [41] é descrito uma série de dificuldades no mapeamento de um sistema codificado em uma linguagem de especificação *Lisp-like* para a linguagem *Promela*. As soluções encontradas estão longe do ideal. Por exemplo, listas foram modeladas através de canais de comunicação, e funções foram codificadas como macros e processos.

Se a nossa abordagem fosse utilizada, a descrição do sistema poderia ser analisada diretamente, as aproximações poderiam ser especificadas sem a necessidade de modificar o código do sistema, além de possibilitar que o usuário experimentasse diferentes tipo de aproximação.

[4] Isto é, uso de aproximações.

10.5 *Verisoft*

Verisoft [39] permite que a implementação de sistemas reativos concorrentes seja diretamente analisada. Esta ferramenta permite a exploração sistemática de mapas de estados de sistemas compostos por diversos processadores concorrentes executando código *C* ou *C++*. *Verisoft* utiliza a técnica de execução livre (Seção 3.5.4) para analisar implementações.

Nesta ferramenta, cada processo do sistema concorrente é mapeado em um processo *UNIX*. A execução do sistema de processos é controlada por um processo externo denominado *scheduler*. Este processo observa as operações realizadas pelos processos do sistema e pode suspender suas execuções. Ao reativar a execução de um determinado processo, o *scheduler* pode explorar uma determinada transição do sistema concorrente. Através da reinicialização do sistema, o *scheduler* pode explorar *traces* diferentes do mapa de estados.

A ferramenta permite a detecção de *deadlocks*, *livelocks* e potenciais *divergências* utilizando marcações de progresso semelhantes as descritas na Seção 5.6.2. Violações de assertivas também podem ser detectadas. A operação *VS_assert(expression)* permite a definição de uma assertiva em um determinado ponto do programa, de forma semelhante a marcação *assert* definida na Seção 5.6.2. Para possibilitar a modelagem do ambiente onde os processo, estão sendo executados, a ferramenta provê a operação *VS_toss(n)*. Esta operação retorna não-deterministicamente um número entre 0 e n.

Como o sistema não armazena nenhuma informação sobre o mapa de estados, o usuário deve especificar a pronfundidade máxima de busca, para evitar que o sistema fique em *loop*. *Partial order methods* (Seção 3.5.5) também são utilizados para reduzir o espaço de busca. Entretanto, não fica claro como a ferramenta pode utilizar efetivamente *partial order methods*, esta não utiliza nenhum mecanismo de análise de ponteiros ou *alias*. Em linguagens como *C* é fundamental realizar uma análise de ponteiros para poder determinar se um conjunto específico de operações é interferente ou não.

10.5.1 Comparação com o nosso sistema

Verisoft possui uma implementação relativamente simples que pode ser adaptada para praticamente qualquer linguagem de programação. Podemos ver a implementação desta ferramenta como um transformador que dado um programa *C*, gera um novo programa contendo suporte a análise, e este código gerado possui uma eficiência semelhante a do código original. Este processo de verificação é mais eficiente que um processo interpretativo como o usado na nossa abordagem. Por outro lado, este tipo de arquitetura possui as suas desvantagens, porque o programa (i.e. código com suporte a análise) precisa ser reinicializado toda vez que uma nova alternativa precisa ser explorada. Esta reinicialização se faz necessária, porque como o sistema não armazena nenhuma informação sobre os estados visitados, não há como realizar um *backtracking* para analizar uma alternativa diferente.

A nossa abordagem possui as seguintes vantagens:

- permite que o usuário defina abstrações (aproximações);
- permite que o usuário verifique propriedades temporais (Seção 5.6.2.4);
- possui métodos de detecção de divergência mais sofisticados do que simplesmente definir uma profundidade máxima de busca(Seção 5.6.2.3);
- para analisarmos alternativas diferentes (*traces* diferentes) o processo não precisa ser reinicializado.

SPEC ⟨ *procedure or method name* ⟩ (⟨ *formal parameter names* ⟩)
MODIFIES ⟨ *list of variables* ⟩
REQUIRES ⟨ *precondition* ⟩
ENSURES ⟨ *postcondition* ⟩

Figura 10.12: Especificação de procedimentos em *ESC*

Acreditamos que não exista a "melhor" abordagem neste caso. Cada uma destas abordagens possui as suas "vantagens" e "desvantagens". Mas é importante notar, que o *Verisoft* se propõe apenas a ser uma ferramenta de verificação de sistemas concorrentes, enquanto que a nossa abordagem se aplica ao desenvolvimento de interpretadores, analisadores e verificadores de código, para diferentes tipos de linguagem de programação e especificação. Além disso, a nossa abordagem pode utilizar resultados de análises (ex.: análise de ponteiros) para tornar os *partial order methods* eficazes.

No futuro iremos implementar um avaliador parcial para linguagem *PAN*, desta maneira, esperamos poder aproximar a abordagem dos analisadores gerados através do nosso "framework" com abordagens não interpretativas, como a do *Verisoft*.

10.6 *Extended Static Checker*

Extended Static Checker [32, 71] é uma ferramenta que detecta erros em tempo de compilação. Como exemplos de erros detectados temos: índices de *arrays* inválidos, *nil dereferences* e *deadlocks*. A funciona de forma semelhante a ferramenta *lint* [50] para *C*. Todos os "erros" detectados devem ser examinados pelo o usuário.

ESC é implementado utilizando tecnologia de verificação de programas através de provadores de teoremas. O programa deve ser anotado com especificações, que são utilizadas por um "gerador de condições de verificação" para produzir fórmulas lógicas que podem ser provadas, se e somente se, o programa esta livre de uma particular classe de erros. Estas fórmulas lógicas são, então, processadas por um provador automático de teoremas denominado *Simplify* [33].

Apesar do processo ter semelhanças com verificação de programas, este está longe deste objetivo. *ESC* não tenta provar que uma programa faz o que realmente deveria fazer. O principal objetivo é detectar certos tipos de erros. Outro detalhe, é que em *ESC* estamos interessados apenas em provas que "falharam", ou seja, as que não poderam ser provadas. As "provas falhas" alertam o usuário para possíveis erros no código.

ESC verifica código *Modula-3*, porém a ferramenta está sendo atualmente adaptada para verificar código *Java*. O sistema realiza apenas verificações intraprocedurais, logo, o usuário deve especificar os procedimentos do programa com pré e pós condições para conseguir resultados satisfatórios. Quando uma chamada de procedimento é encontrada no programa, o sistema verifica se a pré condição é satisfeita, em caso positivo este passa então a assumir que a pós condição é válida. A Figura 10.12 contém o esqueleto de especificação de procedimentos. O termo *MODIFIES* funciona como um *açúcar sintático* para evitar que o usuário especifique na pós condição todas as variáveis que *não* foram modificadas e todas as que foram modificadas. Esta especificação pode ser comparada as marcações que utilizamos em nossa abordagem para melhorar o resultado da análise (Seção 5.7).

10.6.1 Comparação com o nosso sistema

A abordagem de verificação utilizada por *ESC* é extremamente diferente das baseadas em *Verificação de Modelos*, como a nossa. Contudo, todos os tipos de erros detectados por *ESC*, usando um provador de teorema, podem ser detectados, também, usando as técnicas de Verificação de Modelos.

As nossas críticas a abordagem utiliza por *ESC* são:

- Dificilmente a abordagem utilizada no *ESC* pode ser extendida para verificação interprocedural.
- A abordagem não se aplica a verificação de programas contendo funções de alta ordem.
- Apesar de ser dito que a abordagem se aplica a detecção de *deadlocks*, nenhum exemplo é apresentado. É importante lembrar que verificação de programas concorrentes exige verificações não locais.
- Para obter resultados satisfatórios o usuário tem que especificar todos os procedimentos do programa.
- Nenhum exemplo contendo alocação dinâmica de memória é apresentado. Acreditamos que este exemplo seja problemático para o *ESC* devido ao aparecimento de *alias* entre variáveis.

10.7 *SMV*

A ferramenta *SMV* [61] é um *Verificador de Modelos* baseado em *BDDs*, destinada a verificação de *hardware*. Nesta seção, mostraremos por que Verificadores de Modelos baseados em *BDDs* não são um boa alternativa para verificação de *software*. O mapeamento do estado de um programa para uma *array* de bits é o principal problema na verificação de *software* a partir de *BDDs*. O estado de qualquer programa/especificação relativamente complexo não pode ser mapeado em um número pré definido de bits. Para se chegar a esta conclusão basta considerarmos um programa/especificação contendo diversas variáveis e pontos de controle. Além do mais, as transições do programa tem que ser codificadas por fórmulas de lógica proposicional!

É óbvio que, podemos assumir que o estado de um programa pode ser simplicado através da utilização de abstrações que possibilitem o mapeamento do estado para um *array* de bits. Contudo, este fato é verdade apenas se:

- as aproximações levarem os valores das variáveis do programa para conjuntos de valores abstratos finitos e pequenos;
- o programa não possui funções recursivas;
- não há nenhum mecanismo de alocação dinâmica de processos ou memória.

Embora seja possível aproximar programas contendo estes recursos, a quantidade de *bits* necessária para representar um estado apenas é conhecida após a realização de uma análise que constrói o mapa de estados explicitamente. Se o mapa de estados pode ser contruído explicitamente, não faz sentido utilizar uma técnica simbólica (i.e. *BDDs*) para representar implicitamente os mapas de estados.

Em suma, no nosso ponto de vista não faz sentido gerar explicitamente o mapa de estados e em seguida mapeá-lo para um *BDD*, porque se o mapa de estados esta disponível, i.e. ele cabe na memória do computador, é trivial realizar Verificação de Modelos. Mapear uma linguagem de programação/especificação para a linguagem de especificação de uma ferramenta como o *SMV* é uma tarefa quase impossível, estas linguagens são muito pouco expressivas.

10.8 Métodos Convencionais de Análise de Fluxo de Dados

Esta seção tem como função demonstrar a relação dos algoritmos de análise utilizados em nosso "framework" com os algoritmos tradicionalmente utilizados em análise de fluxo de dados. Como já foi mencionado, os nossos analisadores podem ser comparados aos *analisadores iterativos de fluxo de dados* [68, 1]. Estes métodos iterativos são baseados no cálculo do menor ponto fixo de um conjunto de equações recursivas. Entretanto, na literatura é muito comum encontarmos o termo *maximum fixed point*, e esta não é a única diferença de terminologia. O cálculo do ponto fixo é realizado a partir de $\top$ (e não $\bot$), e o operador $\sqcap$ (e não $\sqcup$) é utilizado para "juntar" informações. Fica claro que a estrutura é dual e equivalente a utilizada em nossa abordagem.

As equações recursivas utilizadas no método interativo são construídas a partir do diagrama de fluxo de controle. A estrutura geral das equações é:

$$\begin{aligned} in(B) &= \begin{cases} Init & para\ B = entry \\ \bigsqcup_{P \in Pred(B)} out(P) & caso\ contrário \end{cases} \\ out(B) &= F_B(in(B)) \end{aligned}$$

Onde a função *Pred(B)* retorna os nodos (*basic blocks*) do diagrama de fluxo de controle que possuem arcos direcionados a B. As funções *in* e *out* representam a informação a ser coletada pela análise. Onde in(B) é a informação válida antes de executar o bloco B e out(B) é a informação válida após executar o bloco B. A função F_B é denominada de função de transferência do bloco B. Esta função representa o efeito produzido pela execução do bloco B.

Em geral, os algoritmos de cálculo do menor ponto fixo [5] são mais sofisticados que os utilizados no exemplo da Figura 2.12. Em [68] foi descrito um algoritmo equivalente, em precisão e eficiência, ao nosso algoritmo descrito na Figura 2.17. É muito comum, também, a utilização de técnicas para acelerar a convergência do ponto fixo. Estas técnicas utilizam informações do tipo *def-use chains* [68].

Apesar destas técnicas de aceleração de convergência, vários compiladores utilizam métodos alternativos, por motivos de eficiência, para a realização de análise de fluxo de dados. Estes métodos alternativos são denominados de análise de fluxo de dados baseados em árvore de controle (*control tree based data-flow analysis*). Nestes métodos, os padrões de equações que se repetem são pré processados com o intuito de melhorar a eficiência do analisador. A Figura 10.13 contém, por exemplo, as funções de transferência do comando *if*. A partir desta estrutura do diagrama de fluxo de controle, podemos definir as seguintes relações entre essas funções de transferência:

$$F_{\textit{if-then-else}} = (F_{then} \circ F_{if/Y}) \sqcup (F_{else} \circ F_{if/N})$$

$$\begin{aligned} in(\text{if}) &= in(\text{if-then-else}) \\ in(\text{then}) &= F_{if/Y}(in(\text{if})) \\ in(\text{else}) &= F_{if/N}(in(\text{if})) \end{aligned}$$

É importante observar que a função $F_{\textit{if-then-else}}$ representa o efeito do comando *if* como um todo.

Como em diversas análises (ex.: *available expressions*) é irrelevante a diferença entre $F_{if/Y}$ e $F_{if/N}$. Podemos simplificar a equação da seguinte forma:

$$F_{\textit{if-then-else}} = (F_{then} \circ F_{if}) \sqcup (F_{else} \circ F_{if})$$

[5] Ou se preferir do *maximum fixed point*.

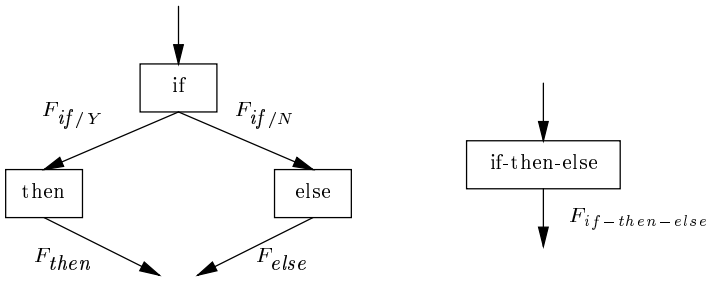

Figura 10.13: Funções de transferência do comando *if*

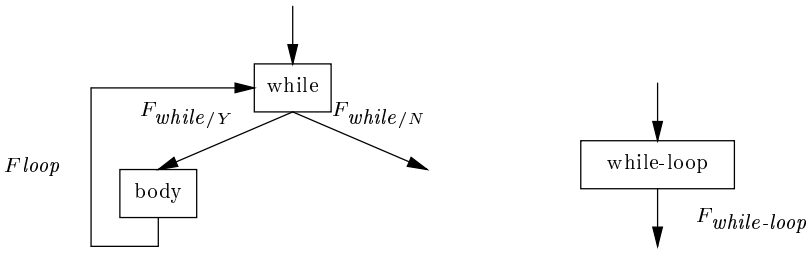

Figura 10.14: Funções de transferência do comando *while*

O comando *while* pode ser tratado de forma semelhante. Para isso, precisamos definir o conceito de *Kleene closure* de uma função. A *Kleene closure* f^* de uma função f é definida como:

$$f^0 = id, n \geq 1, f^n = f \circ f^{n-1}$$

$$\forall x \in L \cdot f^*(x) = \lim_{n \to \infty} (id \sqcup f)^n(x)$$

A Figura 10.14 contém as funções de transferência do comando *while*, podemos, então, definir as seguintes relações entre elas:

$$\begin{aligned} F_{loop} &= (F_{body} \circ F_{while/Y})^* \\ F_{while\text{-}loop} &= F_{while/N} \circ F_{loop} \end{aligned}$$

$$\begin{aligned} in(\text{while}) &= F_{loop}(in(\text{while-loop})) \\ in(\text{body}) &= F_{while/Y}(in(\text{while})) \end{aligned}$$

Enfim, estas equações dizem que o efeito total do comando *while* é equivalente a executar o corpo do *while* zero ou mais vezes até que a condição não seja satisfeita.

As equações do comando *while* podem ser simplificadas. Se observarmos os reticulados que possuem apenas dois valores ($\top$ e $\bot$) [6] a *Kleene closure* de uma função f é $f^* = id \sqcup f$ [7].

[6] Estes reticulados estão associados a análises denominadas de *bit-vector analysis*. Estas análises produzem respostas do tipo "sim" e "não". Como exemplos de analisadores deste tipo temos: *available expressions*, *live variables* e *reaching definitions*.

[7] Este fato pode ser verificado facilmente, se lembrarmos que existem apenas quatro funções $f \in L \to L$, onde L

Portanto, temos que:

$$\begin{aligned} F_{loop} &= (id \sqcup (F_{body} \circ F_{while/Y})) \\ F_{while\text{-}loop} &= F_{while/N} \circ F_{loop} \end{aligned}$$

Com esta simplificação, obtemos um conjunto de equações não recursivo, não sendo necessário utilizarmos um algoritmo para calcular o ponto fixo. Este é o principal motivo porque os analisadores baseados na árvore de controle são mais eficientes que os baseados no método iterativo. Ressaltamos que esta simplificação aplica-se apenas a casos muito particulares, como em diversas análises (ex.: *available expressions*) é irrelevante a diferença entre $F_{while/Y}$ e $F_{while/N}$, podemos simplificar a equação para a seguinte forma:

$$\begin{aligned} F_{loop} &= (id \sqcup (F_{body} \circ F_{while})) \\ F_{while\text{-}loop} &= F_{while} \circ F_{loop} \end{aligned}$$

Ao utilizar estas equações, um analisador baseado na árvore de controle realiza inicialmente uma etapa *bottom-up* para construir uma função de transferência que representa o efeito de um procedimento como um todo. Em seguida é realizada uma etapa *top-down* onde a informação é propagada para as regiões internas do grafo de fluxo de controle.

Considere o diagrama de fluxo de controle da Figura 10.15. A primeira equação construída pela etapa *bottom-up* é para o comando *while*, temos então que:

$$F_{B4a} = F_{B4} \circ (F_{B6} \circ F_{B4})^* = F_{B4} \circ (id \sqcup (F_{B6} \circ F_{B4}))$$

continuando o processo *bottom-up* temos que:

$$\begin{aligned} F_{B3a} &= F_{B5} \circ F_{B4a} \circ F_{B3} \\ F_{B1a} &= (F_{B2} \circ F_{B1}) \sqcup (F_{B3a} \circ F_{B1}) \\ F_{entrya} &= F_{exit} \circ F_{B1a} \circ F_{entry} \end{aligned}$$

Na etapa *top-down*, temos que:

$$\begin{aligned} in(entry) &= Init \\ in(B1a) &= F_{entry}(in(entry)) \\ in(exit) &= F_{B1a}(in(B1a)) \end{aligned}$$

Para o *if-then-else* reduzido para *B1a* temos:

$$\begin{aligned} in(B1) &= in(B1a) \\ in(B2) &= in(B3a) = F_{B1}(in(B1a)) \end{aligned}$$

é um reticulado contendo apenas $\top$ e $\bot$. E destas quatro funções, apenas três são monótonas: a função identidade id (e $id^* = id$), a função constante $const_\bot$ tal que $\forall x \cdot const_\bot(x) = \bot$ (e $const_\bot^* = id$) e a função constante $const_\top$ tal que $\forall x \cdot const_\top(x) = \bot$ (e $const_\top^* = const_\top$). A função f' tal que $f'(\bot) = \top$ e $f'(\top) = \bot$ obviamente não é monotónica.

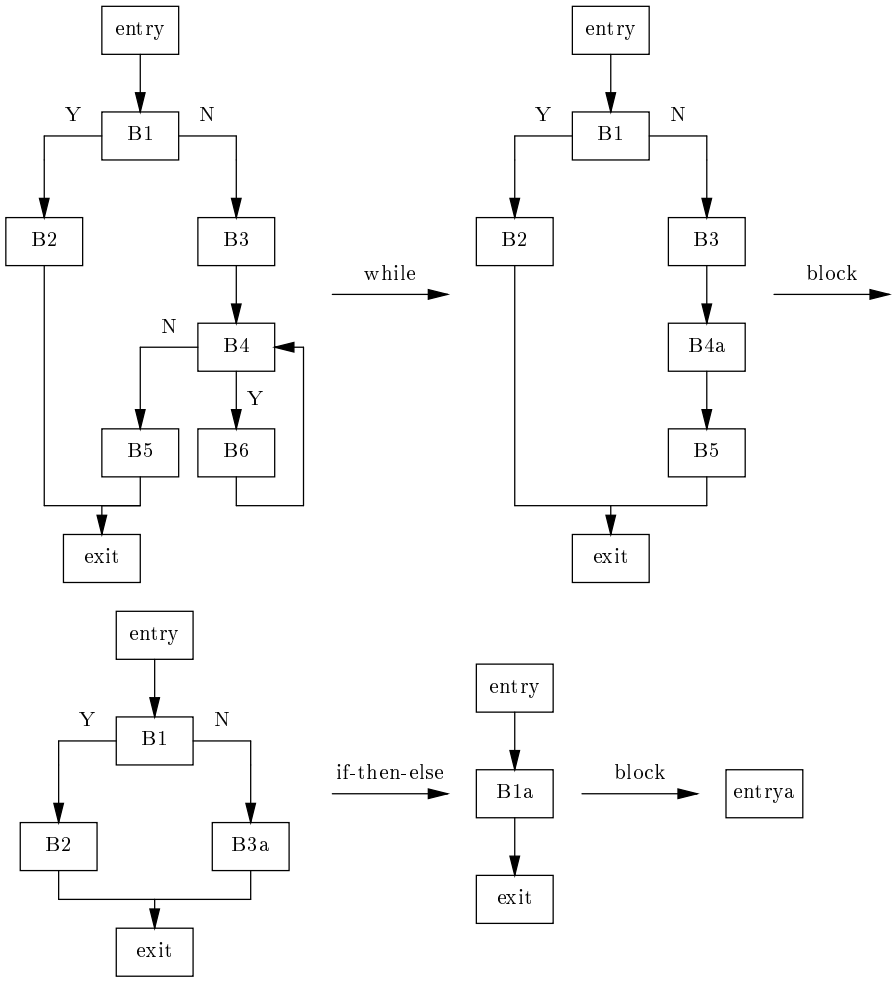

Figura 10.15: *Structural control-flow analysis*

Para o bloco reduzido por $B3a$ temos:

$$\begin{aligned} in(B3) &= in(B1a) \\ in(B4a) &= F_{B3}(in(B3a)) \\ in(B5) &= F_{B4a}(in(B4a)) \end{aligned}$$

Para o comando *while* reduzido por $B4a$ temos:

$$\begin{aligned} in(B4) &= (id \sqcup (F_{B6} \circ F_{B4}))(in(B4a)) \\ in(B6) &= F_{B4}(in(B4)) \end{aligned}$$

O resultado da análise foi obtido sem a necessidade de utilizar um algoritmo de cálculo do menor ponto fixo. Tal procedimento somente foi possível devido a propriedade $f^* = id \sqcup f$ válida para reticulados contendo apenas os elementos $\top$ e $\bot$.

Este algoritmo possui outros detalhes, relativos as chamadas *regiões impróprias*, que não mencionamos, os interessados podem consultar [68].

10.8.1 Comparação com o nosso sistema

Embora estes algoritmos sejam mais eficientes do que os existentes em nossa abordagem, estes possuem os seguintes problemas:

- Não se aplicam a linguagens concorrentes. O método interativo utiliza a noção de *precedente* que não faz muito sentido em uma linguagem concorrente.
- Funcionam apenas para análises intraprocedurais. Para linguagens como *Fortran* e *Cobol* isto não representa um grande problema. Os programas codificados nestas linguagens tendem a ser compostos por uma quantidade pequena de grandes "procedimentos". Além do mais, estas linguagens não possuem alocação dinâmica de memória [8], que complicaria em muito a análise, produzindo resultados muito imprecisos. Por outro lado, em linguagens como *C++* e *Java* os programas tendem a ser compostos por uma quantidade grande de pequenos "procedimentos". Um compilador que utilise apenas análises intraprocedurais não obterá grandes oportunidades de otimização [9].
- Os métodos estruturais, apesar de eficientes, são muito específicos (são linguagem e análise dependentes), e podem ser aplicados somente a análises muito simples.
- As equações estão sujeitas a imprecisões causadas por caminhos impossíveis de serem executados (vide comentário na Seção 2.3). Entretanto, Kindall [56] demostrou que analisadores que utilizam reticulados distributivos não são afetados por este problema.

Concluindo, nossa abordagem troca a eficiência obtida por estes métodos, por flexibilidade e precisão.

10.9 *BANE (Constraint Solving)*

Um analisador baseado em *constraints* [43, 44, 3] possui duas etapas. Na primeira, um conjunto de *constraints* é gerado a partir do código do programa. Esta etapa pode ser vista como a especificação do analisador. Na etapa seguinte, o conjunto de *constraints* é resolvido e o resultado conterá a informação de análise desejada. Esta etapa pode ser vista como a implementação. É importante ressaltar que o algoritmo de resolução de *constraints* é independente de análise.

O analisador de *live-variables* [68] pode ser especificado da seguinte forma utilizando *set constraints*:

- Domínio: Conjunto de variáveis do programa.
- Variáveis: $[\![S]\!]_{in}$ (o conjunto de *live-variables* na entrada do ponto de controle S) e $[\![S]\!]_{out}$ (o conjunto de *live-variables* na entrada do ponto de controle S).
- Constantes: S_{def} (o conjunto de variáveis definidas [10] no ponto de controle S) e S_{use} (o conjunto de variáveis acessadas no ponto de controle S).
- Para cada ponto de controle S são gerados os seguintes *constraints*:

$$\begin{aligned} [\![S]\!]_{in} &= S_{use} \cup ([\![S]\!]_{out} - S_{def}) \\ [\![S]\!]_{out} &= \bigcup_{X \in succ(S)} [\![X]\!]_{in} \end{aligned}$$

[8] Análise de ponteiros "não trivial" é um processo de análise interprocedural.

[9] Este problema pode ser minimizado pela utilização de métodos *inline*.

[10] Na terminalogia da análise de programas, uma váriavel x é dita definida em um ponto do programa, se o seu valor for modificado por uma atribuição ($x = exp$).

A linguagem para definição de *set constraints* pode ser definida como:

$$E ::= 0 \mid \alpha \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid \neg E_1 \mid c(E_1, \ldots, E_n) \mid c_i^{-1}(E_1)$$

onde c representa um construtor, α uma variável e 0 o conjunto vazio. Um sistema de *set constraints* é definido como:

$$\bigwedge_i E_i \subseteq E_i'$$

Uma interpretação para *set constraints* considera que as *set expressions* E denotam subconjuntos de H, o universo de termos de *Herbrand*. Considere que σ é uma função de $V \to 2^H$, onde V é o domínio de variáveis. Então, podemos estender σ para as *set expressions* da seguinte forma:

$$\begin{aligned}
\sigma(0) &= \emptyset \\
\sigma(E_1 \cup E_2) &= \sigma(E_1) \cup \sigma(E_2) \\
\sigma(E_1 \cap E_2) &= \sigma(E_1) \cap \sigma(E_2) \\
\sigma(\neg E) &= H - \sigma(E) \\
\sigma(c(E_1, \ldots, E_n)) &= \{c(t_1, \ldots, t_n) \mid t_i \in \sigma(E_i)\} \\
\sigma(c_i^{-1}(E)) &= \{t_i \mid c(t_1, \ldots, t_n) \in \sigma(E)\}
\end{aligned}$$

Portanto, σ é uma solução dos *constraints* $\bigwedge_i E_i \subseteq E_i'$, se e somente se, $\forall i \cdot \sigma(E_i) \subseteq \sigma(E_i')$.

As projeções c_i^{-1} podem ser utilizadas para modelar seletores de estruturas de dados (ex.: *hd* e *tl*). Além disso, estas podem ser utilizadas para codificar um tipo de implicação:

$$c_1^{-1}(c(A, B)) = \begin{cases} A & \text{se } B \neq 0 \\ 0 & \text{se } B = 0 \end{cases}$$

temos, então, que:

$$B \neq 0 \Rightarrow A \subseteq C \equiv c_1^{-1}(c(A, B)) \subseteq C$$

Para exemplificar o uso de *set contraints*, considere um analisador de classes para linguagens orientadas a objetos não tipadas. Este analisador funciona como um mecanismo de inferência de tipos. Podemos, então, definir o analisador da seguinte forma:

- Uma variável $[\![e]\!]$ para cada expressão e do programa.
- O analisador atribui todas as possíveis classes de e a $[\![e]\!]$.
- Constraints:

$$\begin{aligned}
[\![new\ C]\!] &= \{C\} \\
[\![if\ e_1\ then\ e_2\ else\ e_3]\!] &= [\![e_2]\!] \cup [\![e_3]\!] \\
[\![id := e]\!] &= [\![e]\!] \subseteq [\![id]\!]
\end{aligned}$$

- Para cada aplicação de método $e_0.f(e_1, \ldots, e_n)$ e para cada classe C com método $f(x_1, \ldots, x_n) = e$, temos:

$$C \in [\![e_o]\!] \Rightarrow \bigwedge_i ([\![e_i]\!] \subseteq [\![x_i]\!])$$

O sistema *BANE* [2, 35] implementa diversos algoritmos de resolução de *constraints*, e é utilizado para especificar analisadores de código.

10.9.1 Comparação com o nosso sistema

Na verdade, não estamos comparando os sistemas, mas sim as abordagens. A questão é "qual é a melhor abordagem, Interpretação abstrata ou *Constraint Solving*?". Todavia, esta pergunta não possui uma resposta definitiva. Ambas as abordagens possuem vantagens e desvantagens, das quais podemos citar:

- Em *constraint solving*, o usuário precisa apenas especificar o mapeamento do programa para uma conjunto de *constraints*. O usuário não precisa especificar aproximações. Porém, o uso de aproximações em interpretações abstratas acelera o processo de análise.
- A resolução de *set constraints* é um problema exponencial. Existe uma subcategoria de *set contraints* que pode ser resolvida em tempo $O(n^3)$ (vide Apêndice C).
- Alguns analisadores não podem ser especificados com *set constraints* (ex.: o analisador de *modos* da linguagem *PAN*). Por outro lado, outros analisadores podem ser facilmente codificados (ex.: *live variables*).
- Como os *constraints* estão associados a pontos de controle do programa, estes estão sujeitos as imprecisões causadas por caminhos impossíveis de serem executados.
- A análise de linguagens concorrentes é imprecisa com *set constraints*. Em contra partida, a análise pode servir como uma aproximação inicial (ex.: *flow insensitive pointer analysis*).
- A abordagem baseada em *set constraints* não pode ser utilizada efetivamente para verificar *software*.

Acreditamos, portanto, que as abordagens baseadas em interpretação abstrata e *constraint solving* são na verdade complementares. Um projeto futuro será a integração de nosso sistema com a ferramente *BANE*.

Capítulo 11

Conclusão

11.1 Resultados

Desenvolvimento de um "framework" comum para análise e verificação de programas e especificações.

Em cada capítulo desta tese foram apresentadas as conclusões relativas ao mesmo, e as devidas contribuições. Este capítulo possui a função de ressaltar os principais resultados e contribuições.

O nosso principal resultado foi o desenvolvimento de um "framework" comum para o desenvolvimento modular de analisadores e verificadores de programas e especificações. Estas ferramentas são desenvolvidas em nosso "framework" através de uma abordagem operacional. Todos os tipos de análise e verificação são baseados no conceito de *mapa de estados*. Apesar de estarmos interessados em análise e verificação, a nossa abordagem também permite a construção de interpretadores modulares, que são extremamente úteis para a realização de simulações e desenvolvimento de novas linguagens.

A partir do conceito de mapa de estados, desenvolvemos analisadores mais precisos que os obtidos na abordagem "clássica" de *análise de fluxo de dados*. O conceito de mapa de estados permitiu que realizássemos análises interprocedurais e tratássemos linguagens concorrentes provando, desta forma, a generalidade da nossa abordagem. Também apresentamos algoritmos de diferentes graus de precisão e eficiência.

Os nossos verificadores possuem o objetivo prático de detectar erros. Apresentamos diversas técnicas de verificação que podem ser aplicadas a programas e especificações. Utilizamos aproximações para permitir a verificação de programas com mapas de estados muito grandes ou potencialmente infinitos. Ao contrário dos analisadores, os verificadores não precisam, a princípio, terminar a execução. Mostramos como utilizar verificadores interativamente a partir de conceitos depuração convencionais.

Desenvolvemos a nossa própria linguagem lógica (PAN). A linguagem PAN possui diversos recursos que são fundamentais para a implementação de nossos interpretadores, analisadores e verificadores. A notação *mixfix* provê grande flexibilidade na definição da árvore de sintaxe abstrata e na semântica da linguagem a ser analisada. A interface com a linguagem C, permite a "comunicação" com o mundo "exterior". O suporte a atualizações destrutivas que não prejudicam a interpretação declarativa da linguagem, garantindo a eficiência sem sacrificar a elegância da linguagem. Além disso, a linguagem PAN possui eficiência comparável a obtida com linguagens imperativas. Na implementação do compilador de PAN desenvolvemos analisadores mais sofisticados que os existentes em outros compiladores de linguagens lógicas.

Mostramos motivações para o uso de $DSLs$ no desenvolvimento de software, e como o nosso "framework" de análise e verificação pode ser utilizado para produzir software mais confiável

e eficiente. Dentro deste escopo, mostramos que *DSLs* podem ser vistas como especificações. Mostramos que o uso de "marcações" pode auxiliar os processos de verificação e análise. A linguagem *PAN* também pode ser utilizada na implementação de transformadores e compiladores para estas linguagens. Desta forma, o nosso "framework" é uma solução completa para o desenvolvimento de *DSLs*.

11.1.1 Principais contribuições

- Um "framework" comum para análise e verificação.
- Construção modular de analisadores, verificadores e interpretadores.
- Reutilização de fragmentos de analisadores de código (i.e. *semantic legos*).
- Técnica de verificação utilizando execução livre + aproximações.
- Técnica de verificação utilizando funções de detecção de divergência.
- Técnica para análise interprocedural baseada em contextos de uso.
- Técnicas para análise de linguagens concorrentes.
- Uso de marcadores sofisticados para melhorar o resultado da análise e verificação.
- Linguagem lógica *PAN* que possui sistema de tipos polimórfico, notação *mixfix*, atualizações destrutivas e interface limpa com a linguagem *C*.
- Compilador eficiente para a linguagem *PAN*.
- *DSLs* como especificações que podem ser verificadas, analisadas e transformadas em código eficiente.
- Semântica denotacional × operacional no desenvolvimento de analisadores de código.
- Implementação de *Modular SOS*.

11.2 Trabalhos Futuros

Diversas soluções para o problema de geração de analisadores e verificadores de código foram apresentadas. Foram apontadas diversas direções para trabalhos futuros durante o processo de desenvolvimento deste trabalho.

11.2.1 Avaliador parcial para *PAN*

A primeira extensão para o nosso "framework" será o desenvolvimento de avaliador parcial [51] para a linguagem *PAN*. A implementação não será complexa, visto que já possuimos toda a infraestrutura necessária: analisador sintático, analisadores de *modo* e determinismo. Acreditamos que obteremos resultados muito melhores que os obtidos na literatura de avaliadores parciais de *Prolog* [84], porque temos informação de análise muito mais precisa.

O avaliador parcial irá melhorar a eficiência de nossos analisadores e verificadores. Este avaliador permitirá a geração de verificadores específicos para um determinado programa, dentro do espírito de ferramentas o *Verisoft*.

O uso de avaliadores parciais também é a opção natural para abordagens modulares como a nossa, permitindo a recuperação da eficiência perdida devido a uso de conceitos como encapsulamento de dados e abstrações.

11.2.2 Integração com Ferramentas Externas

Como já foi dito, a integração com outras ferramentas de análise e verificação é um tópico interessante a ser estudado. Estamos particularmente interessandos em interfacear o nosso "framework" com o sistema *BANE* baseado em *constraint solving*. O principal problema para esta integração está no fato de que o nosso "framework" é implementado em *C*, e *BANE* em *SML*.

11.2.3 *Self Application*

Um trabalho futuro interessante é implementar *PAN* utilizando *PAN* e os analisadores construídos em nosso "framework". Esta implementação permitirá o estudo de novas análises que poderão melhorar ainda mais a eficiência do compilador. Dentro deste contexto, seria, também, interessante implementar o avaliador parcial de *PAN* em *PAN*.

11.2.4 Novos analisadores e verificadores

Paralelamente ao desenvolvimento dos trabalhos futuros descritos nas seções anteriores, iremos especificar outras linguagens de programação e domínio. Estamos particularmente interessados em especificar a linguagem *Java*, e a partir desta especificação desenvolver verificadores de código para a mesma. Estes verificadores serão particularmente utéis para a verificação de propriedades de segurança de programas *Java*.

Apêndice A

Teorema de Rice

Seja $\mathcal{C}$ uma coleção de funções parcialmente recursivas de uma variável. Então $\{x \mid \varphi_x \in \mathcal{C}\}$ possui uma função característica recursiva se e somente se $\mathcal{C}$ é vazio ou contem todas as funções parcialmente recursivas.

Notação:

- φ_x representa a função parcialmente recursiva cujo *godël number* é x. Ou seja, x é um número que identifica univocamente a função φ_x.
- $\#f$ "retorna" o *godël number* associado a função f. Logo temos que $\#\varphi_x = x$.

Demonstração:

- Parte 1: Se $\mathcal{C}$ é vazio ou contem todas as funções parcialmente recursivas, então $\{x \mid \varphi_x \in \mathcal{C}\}$ possui uma função característica recursiva.
  - Se $\mathcal{C}$ é vazio, então $\lambda x.0$ é a função característica.
  - Se $\mathcal{C}$ contem todas as funções parcialmente recursivas, então $\lambda x.1$ é a função característica.
- Parte 2: Se $\{x \mid \varphi_x \in \mathcal{C}\}$ possui uma função característica recursiva, então $\mathcal{C}$ é vazio ou contem todas as funções parcialmente recursivas.
  Demonstração:
  Suponha por absurdo que $\mathcal{C}$ não seja vazio e nem contenha todas as funções recursivas. Logo, existe f_1 e f_2, tal que $f_1 \in \mathcal{C}$ e $f_2 \notin \mathcal{C}$. Seja g a função característica de $\{x \mid \varphi_x \in \mathcal{C}\}$, logo temos que $g(\#f_1) = 1$ e $g(\#f_2) = 0$. Seja aux_1 e aux_2 duas funções definidas como:

$aux_1(h, x)$
    $\varphi_h(h)$
    return f_1(x)

$aux_2(h, x)$
    $\varphi_h(h)$
    return f_2(x)

Abstratamente, aux_1 "executa" a função φ_h passando o *godël number* h como parâmetro, e em seguida "retorna" o valor de $f_1(x)$. aux_2 "funciona" de forma semelhante. Para cada h, podemos definir as funções aux_{1h} e aux_{2h} como:

$$aux_{1h} = \varphi_{s(\#aux_1,h)}$$

$$aux_{2h} = \varphi_{s(\#aux_2,h)}$$

Onde s é a função recursiva que "fixa" o primeiro parâmetro de uma outra função, isto é,

$$\forall h \forall x.\varphi_f(h,x) = \varphi_{s(f,h)}(x)$$

Considere agora os seguintes fatos:

- $g(\#f) \neq g(\#w) \Rightarrow f \neq w$
- Se $\varphi_h(h)$ converge, então $\forall x.aux_{1h}(x) = f_1(x)$, i.e. $aux_{1h}(x) = f_1(x)$ [1]. Logo, Se $aux_{1h}(x) \neq f_1(x) \Rightarrow \varphi_h(h)$ diverge. Então,

$$g(\#aux_{1h}) = 0 \Rightarrow aux_{1h} \neq f_1 \Rightarrow \varphi_h(h)\ \textit{diverge}$$

- Se $\varphi_h(h)$ converge, então $\forall x.aux_{2h}(x) = f_2(x)$, i.e. $aux_{2h}(x) = f_2(x)$. Logo, Se $aux_{2h}(x) \neq f_2(x) \Rightarrow \varphi_h(h)$ diverge. Então,

$$g(\#aux_{2h}) = 1 \Rightarrow aux_{2h} \neq f_2 \Rightarrow \varphi_h(h)\ \textit{diverge}$$

- Se $\varphi_h(h)$ diverge, então $aux_{1h} = aux_{2h}$. Logo se $aux_{1h} \neq aux_{2h} \Rightarrow \varphi_h(h)$ converge. Então,

$$g(\#aux_{1h}) \neq g(\#aux_{2h}) \Rightarrow \varphi_h(h)\ \textit{converge}$$

$g(\#aux_{1h})$	$g(\#aux_{2h})$	
1	1	$\varphi_h(h)$ diverge
1	0	$\varphi_h(h)$ converge
0	1	Não acontece
0	0	$\varphi_h(h)$ diverge

Seja *Stop* uma função definida como:

Stop(h)
  if $g(\#aux_{1h}) = 0$ then return 0
  if $g(\#aux_{2h}) = 1$ then return 0
  return 1

Temos, então uma contradição, já que *Stop* é a função característica de $\{x \mid \varphi_x(x)\ converge\}$. Concluímos, assim, a segunda parte da demonstração do teorema de Rice.

[1] Utilizando uma forma *extensional* de igualdade entre funções.

Apêndice B

Processando notação *mixfix*

O analisador sintático da linguagem *PAN* é bastante complicado devido a notação *mixfix*. Devido a esta notação, a sintaxe da linguagem é altamente ambígua e não pode ser tratada pelas técnicas convencionais de análise sintática, tais como: *LaLR*, *LR* ou *LL*.

Para realizar a análise sintática de *PAN*, utilizamos a seguinte técnica. Primeiramente, utilizamos um analisador sintático *LaLR* para transformar o código em um formato que denominamos pré-árvore de sintaxe abstrata. A função do analisador sintático *LaLR* é processar as partes simples do programa. No caso de *PAN*, todas as declarações podem ser processadas facilmente por um analisador sintático *LaLR*. As cláusulas não podem ser processadas. O analisador sintático *LaLR* se limita a tratar os parênteses que aparecem nas cláusulas, e a transformar os elementos de uma cláusula em *tokens*. A clásula do programa descrito na Figura B.1 é transformada na pré árvore de sintaxe abstrata descrita na Figura B.2.

Para transformar as pré árvores de sintaxe abstrata associadas a cada uma das cláusulas do programa, em árvores de sintaxe abstrata legítimas, utilizamos uma técnica de análise sintática baseada no algoritmo de *Tomita* [97]. Este algoritmo funciona como uma extensão dos algoritmos da família *LR*, onde as ambigüidades são tratadas de uma forma especial. No algoritmo de *Tomita*, sempre que ocorre uma ambigüidade *shift-reduce* ou *reduce-reduce*, é criada uma cópia da pilha utilizada pelo analisador sintático. Uma pilha é destruída sempre que esta estiver numa situação de erro sintático. Se ao final do processamento houver mais de uma alternativa válida (i.e. mais de uma pilha válida) um erro de ambigüidade será gerado.

Para adaptarmos este algoritmo as nossas necessidades, devemos observar que a declaração de operadores *mixfix* pode ser vista como uma regra de uma gramática livre de contexto. Por exemplo, podemos ver a declaração:

```
kind stmt.
kind label.
type _ ; _ : stmt -> stmt -> stmt {prec 50 l-assoc}.
type _ -- _ --> : stmt -> label -> stmt -> o {prec 40}.

S1 -- L --> S1'
|-
(S1 ; S2) -- L --> (S1' ; S2).
```

Figura B.1: Programa simples

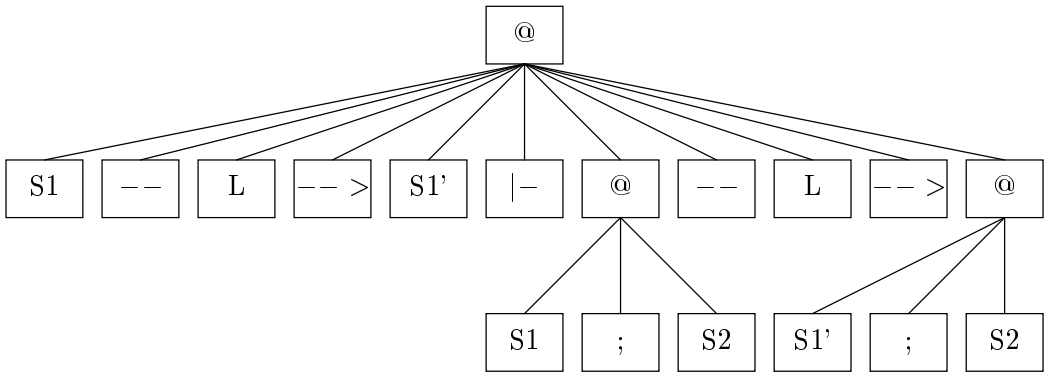

Figura B.2: Pré árvore de sintaxe abstrata

```
type _ ; _ : stmt -> stmt -> stmt
```

como a regra:

```
stmt ::= stmt ; stmt
```

Mas, para aplicarmos o algoritmo de *Tomita*, aparentemente precisamos de uma tabela *LR* ou *LaLR*. Construir uma tabela *LR* ou *LaLR* seria muito custoso, principalmente por que esta teria que ser regerada toda a vez que um programa *PAN* fosse compilado. Por outro lado, podemos ver o uso da tabela *LR* ou *LaLR* no algoritmo de *Tomita* como uma forma de evitar a proliferação de cópias da pilha (alternativas). Como a nossa versão do algoritmo de *Tomita* não utiliza tabela *LR*, a pilha será copiada frequentemente. Entretanto, isso não é um problema grave, visto que a cláusulas são formadas por poucas *tokens* e o uso de parênteses ameniza o problema.

Apesar de não termos uma tabela *LR* para descartar alternativas inválidas, temos as declarações de precedência e o sistema de tipos. Uma redução apenas é realizada, se a regras de precedência e tipagem forem satisfeitas. O nosso algoritmo pode ser sumarizado da seguinte forma:

- Se ainda houver *tokens*, o *shift* pode ser realizado.
- Se o topo da pilha puder ser reduzido, então, verifique se as regras de precedência e tipagem são satisfeitas. Em caso afirmativo, crie uma cópia da pilha, e conseqüentemente uma nova alternativa. Note que, mais de uma alternativa pode ser criada nesse passo.
- Se não houver mais *tokens* para serem consumidas por uma alternativa, a alternativa deve ser reduzida até gerar o termo resultante. Se não for possível, a alternativa é descartada.

Para se otimizar o processo, antes de iniciarmos este, todos os subtermos (os termos entre parênteses) são pré-processados. Este procedimento evita que diferentes alternativas reprocessem o mesmo subtermo. Para otimizarmos a verificação se o topo da pilha é redutível, associamos a cada *token* as declarações que podem reduzí-lo. Em nosso exemplo, o *token* ";" é associado as declarações:

```
type _ ; _ : stmt -> stmt -> stmt.
type _ ; _ : o -> o -> o.
```

Lembre que a segunda declaração é o *ou-lógico*, que é pré-definido em *PAN*. O *token* "--" é associado a declaração:

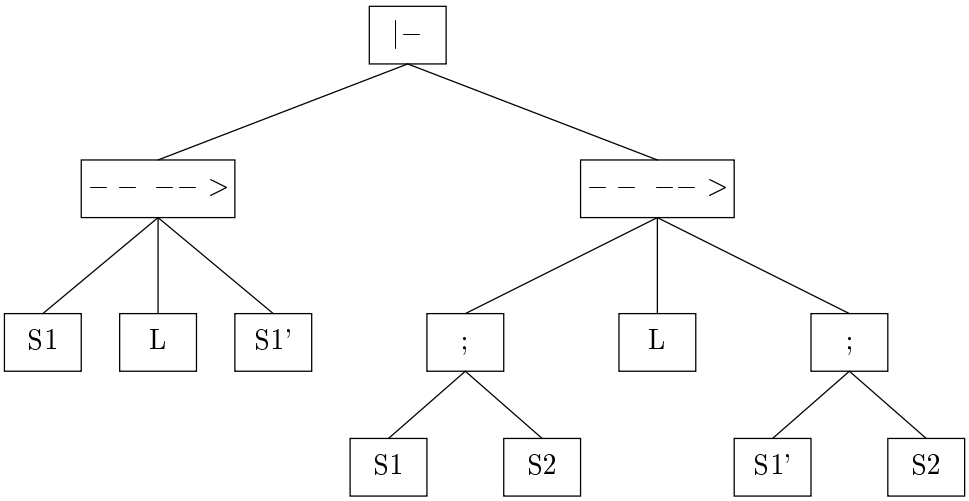

Figura B.3: Árvore de sintaxe abstrata

```
type _ -- _ --> : stmt -> label -> stmt -> o.
```

O token "`-->`" não é associado a nenhuma declaração, já que este não é o primeiro token da declaração acima.

Apêndice C

Constraint Solving

C.1 Introdução

Nesta seção descrevemos um algoritmo para solução de *set contraints*. Como foi descrito na Seção 10.9, a resolução de *set contraints* é um problema exponencial. Logo, o nosso algoritmo resolve apenas uma subcategoria de *set contraints* que pode ser realizada em $O(n^3)$. Esta subcategoria é suficiente para a implementação de um analisador de ponteiros insensível ao fluxo de controle [37]. Esta subcategoria é denominada de *inclusion contraints*.

Inclusion constraints podem ser naturalmente representados por grafos. Por exemplo, os *constraints* $X \subseteq Y \subseteq Z$, podem ser representados por um grafo contendo 3 nodos (X, Y e Z), e os arcos (X, Y) e (Y, Z). O processo de resolução de *constraints* consiste em adicionar novos arcos ao grafo, para representar *constraints* implicados pelo sistema. No exemplo anterior o arco (X, Z) seria adicionado no processo de resolução.

A nossa implementação, e a grande maioria das implementações de *inclusion constraints* [36], representa o grafo descrito acima utilizando listas de adjacência. Por exemplo, o *constraint* $X \subseteq Y$ é representado por um ponteiro para Y na lista de adjacência de X. Dizemos que um arco é predecessor se um ponteiro para X está na lista de adjacência de predecessores de Y. Dizemos que um arco é sucessor se um ponteiro para Y está na lista de adjacência de sucessores de X. Em nossa implementação um arco é exclusivamente sucessor ou predecessor.

C.2 *Inclusion Constraints*

Inclusion constraints são formados por um conjunto de *constraints* com o seguinte formato:

$$L \subseteq R$$

Onde L e R podem ser:

- variáveis
- construtores com um número n de parâmetros
- 0 (conjunto vazio)
- 1 (conjunto universo)

Cada construtor c possui uma assinatura indicando a sua aridade e variância. Um construtor é covariante em um parâmetro, quando o conjunto denotado por $c(\ldots)$ torna-se maior a medida que o parâmetro torna-se maior. Analogamente, um construtor é contravariante em um

parâmetro, quando o conjunto denotado por $c(\ldots)$ torna-se menor a medida que o parâmetro torna-se maior.

Para simplificar a implementação, podemos considerar que 0 (conjunto vazio) e 1 (conjunto universo) são construtores de aridade 0.

C.3 Grafos para a solução de *Inclusion Constraints*

Em nossa implementação, a solução de sistema de *constraints* é um grafo dirigido $G = (V, E)$ fechado segundo a regra de transitividade. Os arcos E representam *constraints atômicos*. Os nodos (vértices) V são variáveis, "fontes" e "destinos". "Fontes" são termos construídos que aparecem a esquerda de uma inclusão. "Destinos" são termos construídos que aparecem a direita de uma inclusão. Um *constraint* é atômico se está em um dos seguintes formatos:

$X \subseteq Y$ variável-variável

$c(\ldots) \subseteq X$ "fonte"-variável

$Y \subseteq c(\ldots)$ variável-"destino"

Para colocarmos um sistema de *constraints* no formato atômico, basta utilizarmos as seguintes regras:

- Remova *constraints* irrelevantes [1], i.e. os seguintes tipos de *constraints* são irrelevantes:
  - $X \subseteq X$
  - $0 \subseteq$ *"qualquer coisa"*
  - *"qualquer coisa"* $\subseteq 1$
- Interrompa o processo e responda que não existe uma solução, se um dos seguintes casos for identificado:
  - $c(\ldots) \subseteq d(\ldots)$ e $c \neq d$
  - $c(\ldots) \subseteq 0$ e $c \neq 0$
  - $1 \subseteq c(\ldots)$ e $c \neq 1$
  - $1 \subseteq 0$

  Note que todos os casos se reduzem ao primeiro, quando consideramos que 0 e 1 são construtores de aridade 0.
- Substitua *constraints* da forma:

$$c(e_1, \ldots, e_n) \subseteq c(e'_1, \ldots e'_n)$$

  pelos constraints:
  - $e_i \subseteq e'_i$ se c é covariante no parâmetro i
  - $e'_i \subseteq e_i$ se c é contravariante no parâmetro i

Em nossa implementação os *constraints* atômicos são representados no grafo da seguinte forma:

[1] Irrelevante no sentido de não restringir nada.

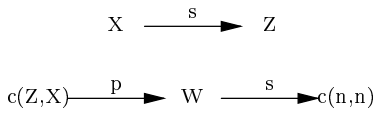

Figura C.1: Grafo inicial

$X \subseteq Y$ arco sucessor (X, Y)

$c(\ldots) \subseteq X$ arco predecessor $(c(\ldots), X)$

$X \subseteq c(\ldots)$ arco sucessor $(X, c(\ldots)$

O propósito desta representação será explicado posteriormente.
Representaremos um arco (X, Y) como:

- $X \xrightarrow{p} Y$ se (X, Y) é predecessor
- $X \xrightarrow{s} Y$ se (X, Y) é sucessor

Novos arcos são adicionados ao grafo, segundo a regra de "fechamento":

$$L \xrightarrow{p} X \xrightarrow{s} R \Rightarrow L \subseteq R$$

Note que esta regra adiciona um novo *constraint* e não um novo arco. O motivo é que $L \subseteq R$ pode não ser um *constraint* atômico. Logo, após a aplicação desta regra, o *constraint* deve ser transformado (se necessário) em um conjunto de *constraints* atômicos, que por sua vez serão transformados em arcos.

A aplicação desta regra propaga os vértices "fonte" a todas as variáveis alcançáveis. A escolha dos arcos predecessores e sucessores é motivada pela implementação da regra de "fechamento". A nossa escolha de arcos "predecessores" e sucessores permite um implementação local desta regra. Isto é, dado uma variável X, basta aplicarmos a regra para todas as combinações de sucessores e predecessores de X.

No exemplo a seguir, assuma que c é um construtor de aridade 2, que é covariante no primeiro argumento e contravariante no segundo argumento, e n é um construtor com aridade 0. Considere agora, o seguinte conjunto de *constraints*:

$$\begin{aligned} W &\subseteq c(n, n) \\ c(X, W) &\subseteq c(Z, c(Z, X)) \end{aligned}$$

Ao transformarmos este sistema no formato atômico, obtemos:

$$\begin{aligned} W &\subseteq c(n, n) \\ X &\subseteq Z \\ c(Z, X) &\subseteq W \end{aligned}$$

Este sistema é representado graficamente na Figura C.1. Neste grafo, podemos então aplicar a regra $c(Z, X) \xrightarrow{p} W \xrightarrow{s} c(n, n)$ e obtermos o *constraint* $c(Z, X) \subseteq c(n, n)$. Simplificando este novo *constraint*, obtemos os seguintes *constraints* atômicos: $Z \subseteq n$ e $n \subseteq X$. A Figura C.2 contem o grafo após a adição dos arcos correspondentes a estes dois novos *constraints*.

Observando o grafo da Figura C.2, fica claro que a regra $n \xrightarrow{p} X \xrightarrow{s} Z$ pode ser aplicada. A Figura C.3 contem o grafo resultante, onde nenhuma arco pode mais ser adicionado. Mostrando desta forma que o sistema de *constraints* possui uma solução.

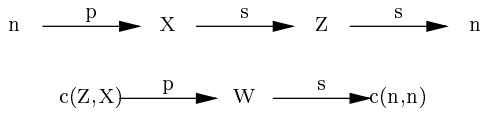

Figura C.2: Grafo após a aplicação da regra $c(Z,X) \xrightarrow{p} W \xrightarrow{s} c(n,n)$

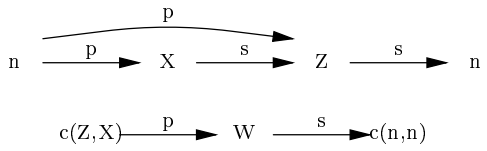

Figura C.3: Grafo após a aplicação da regra $n \xrightarrow{p} X \xrightarrow{s} Z$

C.4 Análise de ponteiros via *inclusion constraints*

Nesta seção mostramos como *inclusion constraints* podem ser utilizados para obtermos informação insensível ao fluxo de controle sobre *aliasing*. Esta abordagem pode ser vista como uma abstração das abordagens propostas por Andersen [5] e Steensgaard [92]. O analisador de Andersen é mais preciso e possui complexidade $O(n^3)$, onde n é o tamanho do programa. A analisador de Steensgaard é menos preciso, mas a complexidade é quase linear $O(n\alpha(n,n))$, onde α é a inversa da função de Ackerman.

Os dois tipos de analisador constroem um grafo contendo informação sobre *points-to*, i.e. que variável aponta para qual. A partir desta informação é trivial obtermos informação sobre *aliasing*. Os nodos do grafo representam posições de memória ou conjuntos de posição de memória. Um arco entre os nodos x e y representa a situação onde uma localização de x aponta para uma localização de y. Posições de memória podem ser variáveis locais e globais, endereços de funções e objetos do *heap*.

A Figura C.5 contem o grafo produzido pelos analisadores de Andersen e Steensgaard, para o programa descrito na Figura C.4.

O nosso prótotipo utiliza uma análise baseada na técnica de Andersen. Informalmente, a análise de Andersen começa com um grafo *points-to*, que é em seguida fechado em relação a regra:

> Para cada atribuição $e_1 = e_2$, qualquer elemento no conjunto *points-to* de e_2 também deve estar no conjunto *points-to* de e_1.

C.4.1 Descrevendo o algoritmo utilizando *inclusion constraints*

O grafo *points-to* de Andersen é composto de localizações abstratas de memória $\{l_1, \ldots, l_n\}$ e variáveis $X_{l_1}, \ldots, X_{l_n}$ representando as localizações apontadas por uma localização abstrata.

```
a = &b;
b = &c;
a = &d;
d = &e;
```

Figura C.4: Programa simples

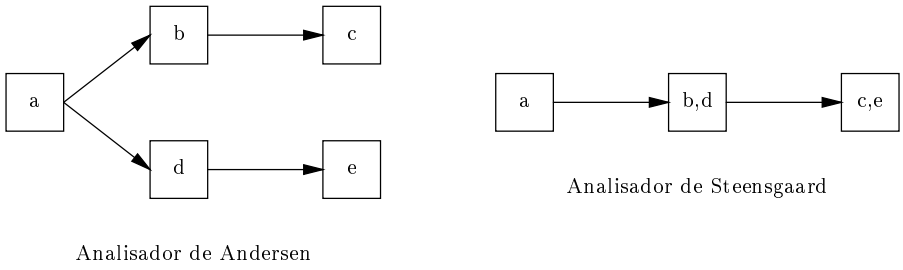

Figura C.5: Grafos gerados pelos algoritmos de Andersen e Steensgaard

Por exemplo, o grafo da Figura C.5 pode ser formulado como:

$$\begin{aligned} X_{l_a} &= \{l_b, l_d\} \\ X_{l_b} &= \{l_c\} \\ X_{l_c} &= \emptyset \\ X_{l_d} &= \{l_e\} \\ X_{l_e} &= \emptyset \end{aligned}$$

A associação entre X_{l_i} e l_i é implícita na formulação de Andersen, resultando, desta maneira, em um algoritmo *ad-hoc* de resolução de *constraints*. Nossa implementação segue a abordagem descrita em [37]. Em nossa implementação, representamos esta associação usando um construtor denominado ref. Logo, $ref(\{l_i\}, X_{l_i})$ representa a associação entre X_{l_i} e l_i. Note que podemos ver $ref(\{l_i\}, X_{l_i})$ como uma espécie de tipo associado a localização abstrata l_x. Usando esta notação, podemos definir indutivamente a regra de geração *constraints* de Andersen como um mecanismo de "inferência de tipos":

- $x : ref(\{l_x\}, X_{l_x})$.
- $\&e : ref(0, t)$ se $e : t$.
- $e_1 = e_2 : t_2$ se $e_1 : t_1$, $e_2 : t_2$ e $t_2 \subseteq t_1$. O significado do *constraint* $t_2 \subseteq t_1$ é que qualquer elemento apontado por t_2 também será apontado por t_1.

Note que não mostramos (propositalmente) o caso da dereferenciação. Entretanto, estas regras são suficientes para resolvermos o exemplo da Figura C.4. Gerando os *constraints* para este exemplo, obtemos:

$$\begin{aligned} ref(0, ref(\{l_b\}, X_{l_b})) &\subseteq ref(\{l_a\}, X_{l_a}) \\ ref(0, ref(\{l_c\}, X_{l_c})) &\subseteq ref(\{l_b\}, X_{l_b}) \\ ref(0, ref(\{l_d\}, X_{l_d})) &\subseteq ref(\{l_a\}, X_{l_a}) \\ ref(0, ref(\{l_e\}, X_{l_e})) &\subseteq ref(\{l_d\}, X_{l_d}) \end{aligned}$$

Colocando no formato atômico, temos:

$$\begin{aligned} ref(\{l_b\}, X_{l_b}) &\subseteq X_{l_a} \\ ref(\{l_c\}, X_{l_c}) &\subseteq X_{l_b} \\ ref(\{l_d\}, X_{l_d}) &\subseteq X_{l_a} \\ ref(\{l_e\}, X_{l_e}) &\subseteq X_{l_d} \end{aligned}$$

Obviamente, esta já é a solução do sistema de *constraints*. Para extrairmos a informação desejada, basta procurarmos por arcos $ref(\{l_i\},\ldots) \stackrel{p}{\rightarrow} X_{l_j}$, que representam o fato $l_i \in X_{l_j}$, ou seja, l_j *pode* apontar para l_i.

Agora, basta mostramos o caso da dereferenciação. Abstratamente, se uma expressão e contem um ponteiro para algum "tipo" t, então $*e$ possui tipo t. Em outras palavras, podemos definir a regra como:

- $*e : T$ se $e : t$ e $t \subseteq ref(1, T)$

Infelizmente esta regra tem um "problema", como podemos verificar aplicando-a ao comando $*x = y$, e produzindo os seguintes constraints:

$$\begin{aligned} ref(\{l_x\}, X_{l_x}) &\subseteq ref(1, T) \\ ref(\{l_y\}, X_{l_y}) &\subseteq T \end{aligned}$$

Colocando no formato atômico, temos:

$$\begin{aligned} X_{l_x} &\subseteq T \\ ref(\{l_y\}, X_{l_y}) &\subseteq T \end{aligned}$$

Obviamente, esta solução não diz "nada". O "problema" é que o tipo T é apenas um limite superior do tipo que estamos querendo restringir. Isto é, a conexão de y para x está "quebrada", já que T é apenas um limite superior. Este "problema" é semelhante ao encontrado com referências atualizáveis. Referências para um tipo α podem ser vistas como um tipo abstrato de dados contendo duas operações: $get : unit \rightarrow \alpha$ e $set : \alpha \rightarrow unit$. Observe que α aparece de forma covariante e contravariante. Este fato sugere que uma modelagem correta de referências deve conter componentes covariantes e contravariantes. Assim, adicionamos ao nosso tipo ref um elemento contravariante. Agora, uma localização abstrata l_x é representada pelo tipo abstrato $ref(\{l_x\}, X_{l_x}, \overline{X_{l_x}})$. A barra sobre o segundo X_{l_x} visa indicar que este é o parâmetro contravariante. Para atualizar uma localização t com um conjunto T, basta adicionarmos o *constraint* $t \subseteq ref(1, 1, \overline{T})$. Assim, obtemos as seguintes (novas) regras para a produção de *constraints*:

- $x : ref(\{l_x\}, X_{l_x}, \overline{X_{l_x}})$.
- $\&e : ref(0, t, \overline{t})$ se $e : t$.
- $*e : T$ se $e : t$, $t \subseteq ref(1, T, \overline{0})$ e T é uma nova variável.
- $e_1 = e_2 : t_2$ se $e_1 : t_1$, $e_2 : t_2$, $t_1 \subseteq ref(1, 1, \overline{T_1})$, $t_2 \subseteq ref(1, T_2, \overline{0})$, $T_2 \subseteq T_1$ e T_1 e T_2 são variáveis novas.

Aplicando o algoritmo acima, no programa:

```
a = &b;
a = &c;
*a = &d;
```

Obtemos o seguintes constraints:

$$\begin{aligned} ref(\{l_a\}, X_{l_a}, \overline{X_{l_a}}) &\subseteq ref(1, 1, \overline{T_1}) \\ ref(0, ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}), \overline{ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}})}) &\subseteq ref(1, T_2, \overline{0}) \end{aligned}$$

$$
\begin{array}{rcl}
T_2 & \subseteq & T_1 \\
ref(\{l_a\}, X_{l_a}, \overline{X_{l_a}}) & \subseteq & ref(1, 1, \overline{T_3}) \\
ref(0, ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}), \overline{ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}})}) & \subseteq & ref(1, T_4, \overline{0}) \\
T_4 & \subseteq & T_3 \\
ref(\{l_a\}, X_{l_a}, \overline{X_{l_a}}) & \subseteq & ref(1, T, \overline{0}) \\
T & \subseteq & ref(1, 1, \overline{T_5}) \\
ref(0, ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}), \overline{ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}})}) & \subseteq & ref(1, T_6, \overline{0}) \\
T_6 & \subseteq & T_5
\end{array}
$$

Colocando no formato atômico, temos:

$$
\begin{array}{rcl}
T1 & \subseteq & X_{l_a} \\
ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) & \subseteq & T_2 \\
T_2 & \subseteq & T_1 \\
T_3 & \subseteq & X_{l_a} \\
ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) & \subseteq & T_4 \\
T_4 & \subseteq & T_3 \\
X_{l_a} & \subseteq & T \\
T & \subseteq & ref(1, 1, \overline{T_5}) \\
ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) & \subseteq & T_6 \\
T_6 & \subseteq & T_5
\end{array}
$$

Usando a regra de fechamento, adicionamos os seguintes *constraints*:

$$
\begin{array}{rcl}
ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) & \subseteq & T_1 \\
ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) & \subseteq & T_3 \\
ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) & \subseteq & T_5
\end{array}
$$

Aplicando a regra novamente, adicionamos os seguintes *constraints*:

$$
\begin{array}{rcl}
ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) & \subseteq & X_{l_a} \\
ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) & \subseteq & X_{l_a}
\end{array}
$$

Aplicando a regra novamente, adicionamos os seguintes *constraints*:

$$
\begin{array}{rcl}
ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) & \subseteq & T \\
ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) & \subseteq & T
\end{array}
$$

Aplicando a regra novamente, obtemos os seguintes *constraints*:

$$
\begin{array}{rcl}
ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) & \subseteq & ref(1, 1, \overline{T_5}) \\
ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) & \subseteq & ref(1, 1, \overline{T_5})
\end{array}
$$

Note que estes *constraints* não estão no formato atômico, ao convertermos os mesmos, adicionamos os seguintes *constraints* atômicos ao grafo:

$$\begin{aligned} T_5 &\subseteq X_{l_b} \\ T_5 &\subseteq X_{l_c} \end{aligned}$$

Usando a regra de fechamento, adicionamos os seguintes *constraints*:

$$\begin{aligned} ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) &\subseteq X_{l_b} \\ ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) &\subseteq X_{l_c} \end{aligned}$$

A solução do sistema de *constraints* é, portanto:

$$\begin{aligned} T1 &\subseteq X_{l_a} \\ ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) &\subseteq T_2 \\ T_2 &\subseteq T_1 \\ T_3 &\subseteq X_{l_a} \\ ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) &\subseteq T_4 \\ T_4 &\subseteq T_3 \\ X_{l_a} &\subseteq T \\ T &\subseteq ref(1, 1, \overline{T_5}) \\ ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) &\subseteq T_6 \\ T_6 &\subseteq T_5 \\ ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) &\subseteq T_1 \\ ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) &\subseteq T_3 \\ ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) &\subseteq T_5 \\ ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) &\subseteq X_{l_a} \\ ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) &\subseteq X_{l_a} \\ ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) &\subseteq T \\ ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) &\subseteq T \\ T_5 &\subseteq X_{l_b} \\ T_5 &\subseteq X_{l_c} \\ ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) &\subseteq X_{l_b} \\ ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) &\subseteq X_{l_c} \end{aligned}$$

A partir desta solução, podemos extrair a seguinte informação sobre *aliasing*:

$$\begin{aligned} ref(\{l_b\}, X_{l_b}, \overline{X_{l_b}}) &\subseteq X_{l_a} \\ ref(\{l_c\}, X_{l_c}, \overline{X_{l_c}}) &\subseteq X_{l_a} \\ ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) &\subseteq X_{l_b} \\ ref(\{l_d\}, X_{l_d}, \overline{X_{l_d}}) &\subseteq X_{l_c} \end{aligned}$$

Ou seja,

$$\begin{aligned} X_{l_a} &= \{l_b, l_c\} \\ X_{l_b} &= \{l_d\} \\ X_{l_c} &= \{l_d\} \end{aligned}$$

Note que não mostramos as regras relacionadas a invocação de funções, e outros recursos. Entretanto, a abordagem pode ser facilmente extendida para estes casos, sem a necessidade de modificar a estrutura da solução, como foi o caso da dereferenciação. O leitor que tenha interesse pode encontrar a definição destas regras em [37].

Apêndice D

Introdução a Co-Indução

D.1 Introdução

Co-indução é uma ferramenta importante para analisarmos algumas estruturas infinitas [40]. Co-indução é o dual da indução. Quando dizemos que algo é indutivamente definido, queremos dizer que este é a menor solução de algum tipo de inequação. Por exemplo, o conjunto dos números naturais $\mathbb{N}$ é a menor solução (ordenada por $\subseteq$) da inequação:

$$\{0\} \cup \{s(x) \mid x \in X\} \subseteq X$$

O princípio indutivo diz que se algum outro conjunto satisfaz a inequação, então este contém o conjunto definido indutivamente. Para provarmos uma propriedade sobre todos os números naturais, precisamos mostrar que o conjunto X dos elementos que satisfazem a propriedade satisfazem a inequação. Como $\mathbb{N}$ é a menor solução, temos $\mathbb{N} \subseteq X$, e conseqüentemente todos os números naturais possuem a propriedade desejada.

Dualmente, um conjunto é co-indutivamente definido se este for a maior solução de algum tipo de inequação. Suponha que para um mapa de estados (árvore) t, escrevemos então $root(t)$ para denotar a raiz da árvore t, e escrevemos $t \rightarrow t'$ para denotar que existe uma transição de estado entre a raiz de t e a raiz de t', i.e., $root(t) \rightarrow root(t')$. O conjunto de computações potencialmente infinitas podem ser definidos como a maior solução da seguinte inequação [1]:

$$X \subseteq \{t \mid \exists t' \cdot t \rightarrow t' \wedge t' \in X\}$$

Analogamente, o princípio co-indutivo diz que se algum outro conjunto satisfaz a inequação, então o conjunto definido co-indutivamente contém ele. Suponha o mapa de estados formado por três nodos $\{t_1, t_2, t_3\}$, e que $t_1 \rightarrow t_2$, $t_2 \rightarrow t_3$ e $t_3 \rightarrow t_1$. Portanto, o conjunto $\{t_1, t_2, t_3\}$ satifaz a inequação, e o mapa de estados formado por estes três nodos é uma computação potencialmente infinita. Ou seja, todo conjunto formado pelos nodos (estados) de um ciclo de um mapa de estados irá satisfazer a inequação, caracterizando dessa forma uma computação infinita.

Um dos principais usos do princípio co-indutivo é na definição de Bissimilaridade [75, 63]. Bissimilaridade em *CCS* [63] é baseada em transições rotuladas (*labelled transitions*). A transição $a \xrightarrow{\alpha} b$ pode ser entendida como se o programa (processo) a realiza uma transição cuja ação observável é α, e o programa resultante é b. Todo o programa produz uma árvore de derivação [2] possivelmente infinita, cujo os nodos são programas e os arcos são as transições rotuladas pelas ações observáveis. Dois programas são bissimilar se estes forem raízes da mesma árvore de

[1] A menor solução desta inequação é o conjunto vazio.

[2] É o equivalente em *CCS* ao nosso mapa de estados.

derivação, onde somente as transições (ações observáveis) são levadas em conta. Formalmente, definimos que a é bissimlar a b ($a \sim b$) como:

$$a \sim b \Leftrightarrow \begin{cases} \text{se } a \xrightarrow{\alpha} a' \text{ então existe } b' \text{ tal que } b \xrightarrow{\alpha} b' \text{ e } a' \sim b' \\ \text{se } b \xrightarrow{\alpha} b' \text{ então existe } a' \text{ tal que } a \xrightarrow{\alpha} a' \text{ e } a' \sim b' \end{cases}$$

D.2 Formalizando Indução e Co-Indução

Seja U algum conjunto e $F : \wp(U) \to \wp(U)$ uma função monótona (i.e. $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$). Indução e Co-Indução são princípios duais que geram conjuntos a partir da menor e maior solução de equações recursivas da forma $X = F(X)$. Para definirmos indução e co-indução formalmente, precisamos de algumas definições.

Definição 6 *Um conjunto $X \subseteq U$ é dito F-fechado (ou F-closed) se e somente se $F(X) \subseteq X$.*

Definição 7 *Um conjunto $X \subseteq U$ é dito F-denso (ou F-dense) se e somente se $X \subseteq F(X)$.*

Definição 8 *Um ponto fixo de F é uma solução da equação $F(X) = X$.*

Definição 9 $\mu X.F(X) \stackrel{\text{def}}{=} \bigcap\{X \mid F(X) \subseteq X\}$

Definição 10 $\nu X.F(X) \stackrel{\text{def}}{=} \bigcup\{X \mid X \subseteq F(X)\}$

Lema 1 *Auxiliar*

1. *$\mu X.F(X)$ é o menor conjunto F-fechado.*
2. *$\nu X.F(X)$ é o maior conjunto F-denso.*

Teorema 5 *(Tarski-Knaster)*

1. *$\mu X.F(X)$ é o menor ponto fixo de F.*
2. *$\nu X.F(X)$ é o maior ponto fixo de F.*

Dizemos que $\mu X.F(X)$, o menor ponto fixo da equação $X = F(X)$, é um conjunto indutivamente definido por F, e analogamente, que $\nu X.F(X)$, o maior ponto fixo da equação $X = F(X)$, é um conjunto co-indutivamente definido por F. Desta maneira, obtemos os princípios duais de prova:

Indução:	$\mu X.F(X) \subseteq X$	se X é F-fechado
Co-Indução:	$X \subseteq \nu X.F(X)$	se X é F-denso

Refazendo o exemplo sobre números naturais, temos que $0 \in U$ e que $s : U \to U$. Definimos a função monótona $F : \wp(U) \to \wp(U)$ como:

$$F(X) \stackrel{\text{def}}{=} \{0\} \cup \{s(x) \mid x \in X\}$$

e o conjunto $\mathbb{N} \stackrel{\text{def}}{=} \mu X.F(X)$, e o princípio de indução associado é que $\mathbb{N} \subseteq X$ se $F(X) \subseteq X$, que é o mesmo que dizer que:

$$\mathbb{N} \subseteq X \text{ se } 0 \in X \text{ e } s(x) \in X \text{ quando } x \in X$$

Refazendo o exemplo sobre computações potencialmente infinitas, definimos $\uparrow \subseteq \mathit{Prog}$ como o conjunto de computações potencialmente infinitas, e a função monótona $F : \wp(\mathit{Prog}) \to \wp(\mathit{Prog})$ como:

$$\begin{array}{rcl} F(X) & \stackrel{\text{def}}{=} & \{t \mid \exists t' \cdot t \to t' \wedge t' \in X\} \\ \uparrow & \stackrel{\text{def}}{=} & \nu X.F(X) \end{array}$$

O princípio de co-indução associado é:

$$\uparrow \text{ é o maior conjunto } F\text{-denso e } \uparrow = F(\uparrow)$$

Apêndice E

Action Semantics

Este capítulo contem a semântica modular de algumas facetas da *Action Notation* [66], codificada na linguagem PAN.

E.1 Módulo Principal

```
module action.

// generic action semantics support
import action-support.

// language specific declarations... i.e. the specification of
// the domains used in the semantic description
import action-language-specific.

// label implementation
import label.

// facets
import action-basic.
import action-functional.
```

E.2 Módulo de Suporte

```
module action-support.

kind action : type.
kind yielder : type.
kind label : type.
kind data-sort : type.
kind operator : type.

// basic label "methods"
type id : label -> o.
type compose-label : label -> label -> label -> o.

type _ -- _ --> _ : action -> label -> action -> o.
```

```
type _ -- _ -->+ _ : action -> label -> action -> o.
type _ -- _ --> _ : yielder -> label -> yielder -> o.
sub data-sort : yielder.

A1 -- L --> A2
|-
A1 -- L -->+ A2.

A1 -- L1 --> A2,  A2 -- L2 -->+ A3, (compose-label L1 L2 L)
|-
A1 -- L -->+ A3.
```

E.3 Módulo Faceta Básica

```
module action-basic.

// ------------ Abstract Syntax Tree ---------------
type _ or _ : action -> action -> action {prec 60 l-assoc}.
type _ and _ : action -> action -> action {prec 70 l-assoc}.
type _ and then _ : action -> action -> action {prec 80 l-assoc}.
type _ trap _ : action -> action -> action {prec 55 l-assoc}.
type indivisibly _ : action -> action {prec 50 l-assoc}.
type unfolding _ : action -> action {prec 50}.
type unfold : action.
type diverge : action.

type fail : action.
type complete : action.
type escape : action.
type commit : action.

type the _ yielded by _ :
 data-sort -> yielder -> yielder {prec 90}.

type op : operator -> list yielder -> yielder.

// ------------- Transition Definition ---------------

// auxiliary transition function
type evaluate-list : list yielder -> list data-sort -> o.

// values added to the AST
type _ @ _ : action -> action -> action {prec 100 l-assoc}.
type completed : action.
type failed : action.
type escaped : action.
type nothing : yielder.

// auxiliar "method"
```

```
type is-terminated : action -> o.
is-terminated completed.
is-terminated failed.
is-terminated escaped.

A1 -- L --> A1'
|-
A1 or A2 -- L --> A1' or A2.

A2 -- L --> A2'
|-
A1 or A2 -- L --> A1 or A2'.

id L
|-
fail -- L --> failed.

id L
|-
completed or A2 -- L --> completed.

id L
|-
A1 or completed -- L --> completed.

id L
|-
escaped or A2 -- L --> escaped.

id L
|-
A1 or escaped -- L --> escaped.

id L
|-
failed or A2 -- L --> A2.

id L
|-
A1 or failed -- L --> A1.

A1 -- L --> A1', (get-commitment L committed)
|-
A1 or A2 -- L --> A1'.

A2 -- L --> A2', (get-commitment L committed)
|-
A1 or A2 -- L --> A2'.
```

```
id L, (set-commitment L committed L')
|-
commit -- L' --> completed.

A1 -- L --> A1'
|-
A1 and A2 -- L --> A1' and A2.

A2 -- L --> A2'
|-
A1 and A2 -- L --> A1 and A2'.

id L
|-
complete -- L --> completed.

id L
|-
completed and completed -- L --> completed.

id L
|-
escaped and A2 -- L --> escaped.

id L
|-
A1 and escaped -- L --> escaped.

id L
|-
failed and A2 -- L --> failed.

id L
|-
A1 and failed -- L --> failed.

A -- L -->+ T, (is-terminated T)
|-
indivisibly A -- L --> T.

A1 -- L --> A1'
|-
A1 and then A2 -- L --> A1' and then A2.

A2 -- L --> A2'
|-
completed and then A2 -- L --> completed and then A2'.

id L
```

```
|-
completed and then completed -- L --> completed.

id L
|-
completed and then escaped -- L --> escaped.

id L
|-
completed and then failed -- L --> failed.

id L
|-
escaped and then A2 -- L --> escaped.

id L
|-
failed and then A2 -- L --> failed.

A1 -- L --> A1'
|-
A1 trap A2 -- L --> A1' trap A2.

id L
|-
escape -- L --> escaped.

id L
|-
escape trap A -- L --> A.

id L
|-
completed trap A -- L --> completed.

id L
|-
failed trap A2 -- L --> failed.

id L
|-
unfolding A -- L --> A @ A.

id L, is-terminated T
|-
T @ A -- L --> T.

(set-unfolding L A0 L'), A -- L' --> A'
|-
```

```
A @ A0 -- L --> A' @ A0.

(get-unfolding L A0)
|-
unfold -- L --> A0.

id L
|-
diverge -- L --> diverge.

var D : data-sort.
var Ds : data-sort.

id L,
(Y -- L --> (inj-data-sort-yielder D)),
(is-sub D Ds)
|-
the Ds yielded by Y -- L --> D.

id L,
Y -- L --> (inj-data-sort-yielder D),
not (is-sub D Ds)
|-
the Ds yielded by Y -- L --> nothing.

id L, Y -- L --> nothing
|-
the Ds yielded by Y -- L --> nothing.

evaluate-list nil nil.

Y -- L --> (inj-data-sort-yielder Ds),
id L, (evaluate-list YList DsList)
|-
evaluate-list (Y :: YList) (Ds :: DsList).

var DsResult : data-sort.

(evaluate-list YList DsList),
(apply-op OP DsList DsResult), id L
|-
(op OP YList) -- L --> DsResult.
```

E.4 Módulo Faceta Funcional

```
module action-functional.

// ------------ Abstract Syntax Tree ---------------
type give : yielder -> action {prec 50}.
```

```
type choose : yielder -> action {prec 50}.
type check : yielder -> action {prec 50}.
type regive: action.
type _ then _ : action -> action -> action {prec 80 l-assoc}.
type escape with _ : yielder -> action {prec 50}.

type it : yielder.
type them : yielder.
type given : data-sort -> yielder.
type given _ # _ : data-sort -> int -> yielder.

type boolean : data-sort.
type true-value : data-sort.
type false-value : data-sort.
is-sub true-value boolean.
is-sub false-value boolean.

type _ /\ _ : data-sort -> data-sort -> data-sort.

// ------------- Transition Definition ---------------

// values added to the AST
type gave : data-sort -> action {prec 50}.
type escape-gave : data-sort -> action {prec 50}.
type none : data-sort.

// auxiliar "methods"
type concat-data-sorts :
 data-sort -> data-sort -> data-sort -> o.
type get-nth : data-sort -> int -> data-sort -> o.

concat-data-sorts D1 D2 (D1 /\ D2) :-
  not (D1 = (D3 /\ D4) : data-sort).
concat-data-sorts (D1 /\ D2) D3 (D1 /\ D4) :-
  concat-data-sorts D2 D3 D4.

get-nth D N none :- not (D = (D1 /\ D2) : data-sort).
get-nth (D1 /\ D2) N D :-
if N = 1 then D = D1 else N1 is N-1, (get-nth D2 N1 D).

type is-datum : data-sort -> o.
is-datum D :- not (D = (D1 /\ D2) : data-sort).

type is-completed : action -> o.
is-completed completed.
is-completed (gave D).

type is-escaped : action -> o.
is-escaped escaped.
```

```
is-escaped (escape-gave D).

// extending auxiliar method "is-terminated"
is-terminated (gave D).
is-terminated (escape-gave D).

var D : data-sort.
var Ds : data-sort.

id L, Y -- L --> (inj-data-sort-yielder D)
|-
give Y -- L --> gave D.

id L, Y -- L --> nothing
|-
give Y -- L --> failed.

id L, (get-data L D)
|-
regive -- L --> gave D.

id L, Y -- L --> Ds, (is-sub D Ds)
|-
choose Y -- L --> gave D.

id L, Y -- L --> nothing
|-
choose Y -- L --> failed.

id L, Y -- L --> true-value
|-
check Y -- L --> gave(none).

id L, (Y -- L --> false-value; Y -- L --> nothing)
|-
check Y -- L --> failed.

id L
|-
(gave D) or A2 -- L --> gave D.

id L
|-
A1 or (gave D) -- L --> gave D.

id L
|-
(escape-gave D) or A2 -- L --> escape-gave D.
```

```
id L
|-
A1 or (escape-gave D) -- L --> escape-gave D.

id L
|-
(escape-gave D) and A2 -- L --> escape-gave D.

id L
|-
A1 and (escape-gave D) -- L --> escape-gave D.

id L
|-
(escape-gave D) and then A2 -- L --> escape-gave D.

id L, (is-completed C)
|-
C and then (escape-gave D) -- L --> escape-gave D.

id L, concat-data-sorts D1 D2 D
|-
(gave D1) and (gave D2) -- L --> gave D.

id L, concat-data-sorts D1 D2 D
|-
(gave D1) and then (gave D2) -- L --> gave D.

A1 -- L --> A1'
|-
A1 then A2 -- L --> A1' then A2.

set-data L D1 L', A2 -- L' --> A2'
|-
(gave D1) then A2 -- L --> A2'.

(is-completed C1), (is-escaped E2), id L
|-
C1 then E2 -- L --> E2.

(is-escaped E1), id L
|-
E1 then A2 -- L --> E1.

(is-completed C1), id L
|-
C1 then failed -- L --> failed.

id L
```

```
|-
failed then A2 -- L --> failed.

id L, Y -- L --> D
|-
escape with Y -- L --> escape-gave D.

id L, Y -- L --> nothing
|-
escape with Y -- L --> failed.

(set-data L D1 L'), A2 -- L' --> A2'
|-
(escape-gave D1) trap A2 -- L --> A2'.

(is-escaped E1), (is-terminated T2), id L
|-
E1 trap T2 -- L --> T2.

(is-completed C1), id L
|-
(gave D) trap A2 -- L --> (gave D).

id L, (get-data L D), (is-datum D)
|-
it -- L --> D.

id L, (get-data L D)
|-
them -- L --> D.

id L, (get-data L D), (is-sub D Ds)
|-
(given Ds) -- L --> D.

var D' : data-sort.

id L, (get-data L D), (get-nth D P D'), (is-sub D' Ds)
|-
(given Ds # P) -- L --> D'.
```

E.5 Módulo Label

```
module label.

// implementation of the basic label methods
id L. // temporary definition
compose-label L L L. // temporary definition
```

```
type mk-label : label. // temporary definition

kind commitment-flag : type.
type committed : commitment-flag.
type non-committed : commitment-flag.

type get-commitment : label -> commitment-flag -> o.
type set-commitment : label -> commitment-flag -> label -> o.

get-commitment L committed. // temporary definition
set-commitment L C L. // temporary definition


type set-unfolding : label -> action -> label -> o.
type get-unfolding : label -> action -> o.

get-unfolding L A. // temporary definition
set-unfolding L A L. // temporary definition

type get-data : label -> data-sort -> o.
type set-data : label -> data-sort -> label -> o.

get-data L D. // temporary definition
set-data L D L. // temporary definition
```

Bibliografia

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

[2] A. Aiken, M. Fahndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proc. of the 2nd Workshop on Types in Compilation (TIC), Japan*, Mar. 1998.

[3] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[4] H. Ait-Kaci. *Warren's abstract machine: a tutorial reconstruction.* MIT Press, 1991.

[5] L. Andersen. Program analysis and specialization for the C programming language. Ph.d. thesis, DIKU, University of Cophenhagen, May 1994.

[6] S. Anderson and K. Tourlas. Design for proof: An approach to the design of domain-specific languages. *Formal Aspects of Computing*, 10(5&6):452–468, 1998.

[7] A. Arnold. *Finite Transition Systems.* Prentice-Hall, 1992.

[8] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge Press, 1990.

[9] I. D. Baxter. Design Maintenance Systems. *Comm. of ACM*, 35(4):73–89, Apr. 1992.

[10] J. Bell. Software Design for Reliability and Reuse: A proof-of-concept demonstration. In *Tri-Ada'94*, pages 396–404. ACM Press, Nov. 1994.

[11] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *PLILP'90*, volume 456 of *LNCS*, pages 307–323. Springer-Verlag, 1990.

[12] F. Bourdoncle. Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite. Ph.d. thesis, Ecole Polytechnique, 1992.

[13] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications.* Springer-Verlag, 1993.

[14] C. Braga. Modular SOS and Componentware. Not yet published, 1999.

[15] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimization of Prolog programs. In *Proc. of the 1987 International Symposium on Logic Programming, San Francisco, California*, pages 192–204. IEEE Computer Society Press, Aug. 1987.

[16] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[17] S. Chao and B. Bryant. Denotational semantics for program analysis. *ACM SIGPLAN Not.*, 23(1):83–91, Jan. 1988.

[18] E. M. Clarke and E. Emerson. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Programm.*, 2:41–66, 1982.

[19] E. M. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[20] E. M. Clarke, O. Grumberg, and D. E. Long. Verification tools for finite-state concurrent systems. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *LNCS*, pages 124–175. Springer-Verlag, 1993.

[21] E. M. Clarke and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. Assoc. Comput. Mach.*, 33:51–78, 1986.

[22] E. M. Clarke and A. P. Sistla. Deciding full branching time logic. *Information and Control*, 61:175–201, 1984.

[23] P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. In *Proc. of the 12th International Conference of Logic Programming, Kanagawa, Japan*, pages 317–331, June 1995.

[24] J. Cordy and I. Carmichael. The TXL programming language syntax and informal semantics. Technical report, Queen's University at Kinkston, Canada, 1993.

[25] P. Cousot. Abstract Interpretation. *ACM Comp. Surv.*, 28(2):324–328, June 1996.

[26] P. Cousot. Types as Abstract Interpretation. In *POPL 97, Paris, France*, pages 316–331, 1997.

[27] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, Jan. 1977.

[28] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[29] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP'92*, volume 631 of *LNCS*, pages 269–295. Springer-Verlag, 1992.

[30] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[31] J. de Bakker and E. de Vink. *Control Flow Semantics*. MIT Press, 1996.

[32] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. SRC Research Report 159, Compaq Systems Research Center, Dec. 1998.

[33] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: the ESC theorem prover. Src research report (draft), Digital Systems Research Center, Nov. 1996.

[34] V. Donzeau-Gouge. Denotational definitions of properties of program computations. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 11, pages 343–379. Prentice-Hall, 1981.

[35] M. Fahndrich and A. Aiken. Program analysis using mixed term and set constraints. In *Proc. of the 4th International Static Analysis Symposium*, 1997.

[36] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *ACM SIGPLAN'98 Conference on Programming Language Design and Implementation, Montreal, Canada*, 1998.

[37] J. S. Foster, M. Fähndrich, and A. Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report UCB/CSD-97-964, University of California at Berkeley, Aug. 1997.

[38] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion*, volume 1032 of *LNCS*. Springer-Verlag, Jan. 1996.

[39] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Jan. 1997.

[40] A. D. Gordon. A tutorial on co-induction and functional programming. In *Proc. of the 1994 Glasgow Workshop on Functional Programming*, pages 78–95. Springer Workshops in Computing, 1995.

[41] K. Havelind, M. Lowry, and J. Penix. Formal analysis of a space craft controller using Spin. Technical report, NASA, Ames Research Center, California, 1997.

[42] R. C. Haygood. Native code compilation in SICStus Prolog. In *Proc. of the 11th International Conference of Logic Programming, Santa Margherita Ligure, Italy*, pages 190–204, June 1994.

[43] N. Heintze. Set based program analysis. Ph.d. thesis, Carnegie Mellon University, Department of Computer Science, Oct. 1992.

[44] N. Heintze. Set based analysis of ML programs. In *Proc. of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–317, June 1994.

[45] T. A. Henzinger. Some myths about formal verification. *ACM Computing Surveys*, 28 A(4), Dec. 1996.

[46] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[47] G. J. Holzmann. The model checker Spin. *IEEE Trans. Software Engrg.*, 23(5), May 1997.

[48] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Comp. Surv.*, 28(4), 1996.

[49] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell: a nonstrict, purely functional language, version 1.2. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, Mar. 1992.

[50] S. C. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ, 1978.

[51] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[52] N. D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. Tech. Rep., DIKU, Univ. of Copenhagen, Denmark, 1989.

[53] S. P. Jones, N. Ramsey, and F. Reig. C−−: a portable assembly language that supports garbage collection. In *Proc. of the PPDP'99*, 1999.

[54] P. Jouvelot. Semantic Parallelization: a practical execise in abstract interpretation. In *14th POPL, Munich, West Germany*, pages 39–48, Jan. 1987.

[55] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[56] G. A. Kildall. A unified approach to global program optimization. In *POPL 73, Boston, MA*, pages 194–206, Oct. 1973.

[57] D. E. Knuth. Notes on the construction of priority queues: An instructive use of recursion. Informally distributed course notes, Mar. 1977.

[58] J. C. Leite, M. Sant'anna, and F. G. Freita. Draco-PUC: a technology assembly for domain oriented software development. In *Proc. of the 3rd IEEE International Conference of Software Reuse*, 1994.

[59] J. R. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., second edition, 1991.

[60] R. W. Lewis. Programming industrial control systems using IEC 1131-3. Technical report, Control Engineering. The Institution of Electrical Engineers (IEE), London, 1995.

[61] K. L. McMillan. The SMV system. Technical report, Carnegie-Mellon University http://www.cs.cmu.edu/ modelcheck, Feb. 1992.

[62] R. Milner. Theory of type polymorphism in programming. *J. of Computer and System Sciences*, 17(3):348–375, 1978.

[63] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[64] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers and MIT Press, 1990.

[65] E. Moggi. Computational lambda-calculus and monads. In *Proc. 4th Annual Symposium on Logic in Computer Science, Asilomar CA.* IEEE Computer Society Press, 1989.

[66] P. D. Mosses. *Action Semantics.* Cambridge Press, 1992.

[67] P. D. Mosses. Semantics, modularity and rewriting logic. In *WRLA'98, Second International Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, Frace*, Sept. 1998.

[68] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[69] J. M. Neighbors. The Draco approach for constructing software from components. *IEEE Transactions on Software Engineering*, 10(5):564–574, 1984.

[70] J. M. Neighbors. A method for engineering reusable software systems. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, pages 295–320. ACM Press, New York, 1989.

[71] G. Nelson. Techniques for program verification. Ph.d. thesis, Stanford University, 1981.

[72] M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains - part 1. *Theoretical Computer Science*, 13:85–108, 1981.

[73] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[74] F. Nielson. Two-level semantics and abstract interpretation. *Theor. Comp. Sci.*, 69:117–242, 1989.

[75] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science: 5th GI-Conference*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.

[76] T. Parr and J. Lilley. ANTLR 2.10. Reference manual, MageLang Institute, http://www.MageLang.com, 1998.

[77] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[78] G. D. Plotkin. A Structural approach to Operational Semantics. Technical Report FN-19, DAIMI, University of Aarhus, Denmark, Sept. 1981.

[79] G. Polya. *Mathematics And Plausible Reasoning.* Princeton Univeristy Press, 1954.

[80] V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15:33–71, 1986.

[81] J. H. Reif. Data flow analysis of distributed communicating processes. Technical Report TR-12-83, Harvard University, Sept. 1984.

[82] P. L. V. Roy. Can logic programming execute as fast as imperative programming? Ph.d. thesis, UC Berkeley, Dec. 1994.

[83] P. L. V. Roy and A. Despain. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer*, 25(1):54–68, Jan. 1992.

[84] D. Sahlin. An automatic partial evaluator for full Prolog. Ph.D. thesis TRITA-TCS-9101, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991.

[85] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[86] D. A. Schmidt. Data-Flow Analysis is Model Checking of Abstract Interpretations. In *Proc. 25th ACM Symp. on Principles of Prog. Languages.* ACM Press, 1998.

[87] D. A. Schmidt. Trace-based abstract interpretation of operational semantics. *J. Lisp and Symbolic Computation*, 10(3):237–271, 1998.

[88] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[89] O. Shivers. Control flow analysis in Scheme. In *SIGPLAN '88 Conference on PLDI, Atlanta, Georgia*, volume 23(7) of *ACM SIGPLAN Not.*, pages 164–174, July 1988.

[90] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages.* Addison-Wesley, 1995.

[91] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[92] B. Steensgaard. Points-to analysis in almost linear time. Technical Report MSR-TR-95-08, Microsoft Research, 1995.

[93] B. Steffen. Generating Data-Flow Analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115–139, 1993.

[94] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, second edition, 1994.

[95] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic and Computer Science*, volume 2, pages 447–563. Oxford Press, 1992.

[96] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[97] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer, 1985.

[98] G. A. Venkatesh and C. N. Fischer. Spare: A development environment for program analysis algorithms. *IEEE Trans. Software Engrg.*, 18(4), Apr. 1992.

[99] J. Vollmer. Data flow analysis of parallel programs. Interner Bericht 19/95, Universität Karlsruhe, Fakultät für Informatik, Mar. 1995.

[100] P. Wadler. Comprehending monads. In *Proc. of 1990 ACM Conference on Lisp and Functional Programming*, 1990.

[101] P. Wadler. Linear types can change the world! In *Proc. of the IFIP TC2 Conference on Programming Concepts and Methods*, pages 547–566, Apr. 1990.

[102] D. H. D. Warren. Logic programming and compiler writing. *Software-Practice and Experience*, 10(2):97–125, 1980.

[103] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[104] M. Wolfe and H. Srinivasan. Data structures for optimizing programs with explicit parallelism. In H. Zima, editor, *Parallel Computing, 1. Int. ACPC Conference Salzburg, Austria*, number 591 in LNCS, pages 139–156. Springer-Verlag, Sept. 1991.