Este exemplar corresponde à redação final da tese defendida por Silvio Roberto Medeiros Evangelista e aprovada pela Comissão Julgadora em 13 11 92.

Orientador

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica
Departamento de Engenharia de Computação
e Automação Industrial

Abstração Funcional de Programas:

Uma Contribuição ao Entendimento do Código Fonte de um Programa

Silvio Roberto Medeiros Evangelista

Orientadora: Profa. Dra. Beatriz Mascia Daltrini

Co-Orientador: Dr. Fuad Gattaz Sobrinho

Dissertação de mestrado apresentada à Faculdade de Engenharia Elétrica UNICAMP.

Campinas, Novembro de 1992.

Agradecimentos

Agradeço à Empresa Brasileira de Pesquisa Agropecuária (EMBRAPA) pelo apoio, incentivo e suporte financeiro, fundamentais à realização deste trabalho.

A Profa. Dra. Beatriz Mascia Daltrini pela confiança, pela paciência, pelo encorajamento e pela dedicação, que foram fundamentais à conclusão deste trabalho.

Ao Dr. Fuad Gattaz Sobrinho, a quem estou profundamente grato pelas sugestões e pelo auxílio na definição do tema proposto.

Ao Prof. Dr. Mário Jino pelas sugestões e pelas horas de seu trabalho que me foram dedicadas.

Ao Prof. Dr. Adilson Marques da Cunha pelas importantes críticas e sugestões acerca deste trabalho.

As amigas Marcélia Rezende, Maria Fernanda Moura e Sônia Ternes pelo incentivo pessoal e pelo trabalho de implementação da Arvore "And-Or" e do algoritmo "Slicer".

Aos demais amigos do NTIA pelo apoio e companheirismo constantes.

Agradeço, finalmente, à minha querida esposa Adriana pelo imenso carinho, apoio e, também, pelas exaustivas revisões realizadas neste texto.

à minha querida esposa Adriana

Resumo

Este trabalho tem por objetivo desenvolver e implementar um modelo para abstração funcional de programas, a qual é definida nesta pesquisa como a precisa determinação do efeito de um programa sobre as suas variáveis em todas as situações possíveis.

O modelo de abstração funcional proposto é fundamentado na segmentação do programa alvo em termos de suas variáveis relevantes para abstração, na sua decomposição em primos, na simplificação algébrica dos comandos de decisão do programa, na execução simbólica e no uso das técnicas de "Trace-Table" e de resolução das relações de recorrência em iterações.

Este modelo de abstração é validado e consolidado através da implementação de uma ferramenta que utiliza o código fonte como única fonte de informação, gera uma forma intermediária do programa para facilitar a atividade de abstração, segmenta o programa em função de suas variáveis relevantes para abstração, decompõe o segmento encontrado em programas primos e sintetiza cada um dos primos.

Este trabalho contribuiu para a definição de um modelo de abstração funcional promissor em relação ao seu potencial de automatização, bem como à sua aplicabilidade.

índice

Lista de Figuras

Lista de Tabelas

Capítulo 1
Introdução

Este capítulo apresenta a principal motivação do tema da pesquisa, os principais objetivos a serem alcançados e a organização dos demais capítulos desta dissertação.

1.1 Contexto da Pesquisa

Um dos principais fatores que influenciam a chamada crise do software é a atividade de manutenção. A manutenção pode ser definida como a modificação de um produto de software na fase de operação para correção de falhas, para aumento de desempenho ou outros atributos e para adaptação do produto a outros ambientes de hardware ou software [IEEE-83].

A atividade de manutenção de um software é responsável por uma média de 70% de todo o gasto relativo à sua construção [SCAN-90, LEHN-80]. Neste sentido, cabe indagar por que a atividade de manutenção é tão dispendiosa em relação às demais atividades do ciclo de vida de um software.

Um primeiro aspecto a ser considerado numa resposta a esta questão é a existência de um grande acervo de software mal estruturados e, portanto, difíceis de serem mantidos. Observa-se que este acervo é calculado em cerca de 300 bilhões de dólares somente nos Estados Unidos[YEH-90], sendo que o gasto anual em manutenção, no mesmo país, é de 10 bilhões de dólares [CORB-89].

Um segundo aspecto é a própria área de aplicação desta atividade, que envolve "entender e documentar um software existente; estender a sua funcionalidade; adicionar novas funções ao programa; achar e corrigir erros; responder perguntas sobre o software aos usuários e pessoal de operação; reescrever, reestruturar e converter um software; finalmente, gerenciar a sua operação" [CORB-89].

Um terceiro aspecto a ser considerado é o fato do pessoal técnico responsável pela manutenção não ser formado, em geral, pelos analistas e/ou programadores originais do software, o que dificulta a tarefa de entendimento do software(1), a qual é fundamental para o desenvolvimento efetivo das aplicações listadas por Corbi.

---

(1) Estudos mostram que mais de 50% do tempo dos programadores de manutenção é dedicado à tarefa de entendimento do software [CORB-89].

A tarefa de entendimento do software envolve três fontes complementares de informação: leitura da documentação, leitura do código fonte e ganho de conhecimento a partir dos resultados produzidos pelo programa. Entretanto, a documentação pode estar incompleta, não atualizada ou, ainda, não existir. Neste caso, a única fonte de informação realmente confiável é o código fonte [BYRN-91].

O entendimento de um software a partir de seu código fonte é sensivelmente mais demorado do que o entendimento derivado da documentação [CORB-89]. Neste contexto, deve-se ressaltar a importância da engenharia reversa para a tarefa do entendimento, já que ela é a parte do processo de manutenção que auxilia o programador a entender um software, tornando factíveis as mudanças requeridas [CHIK-90].

As técnicas e ferramentas utilizadas pela engenharia reversa podem ser divididas em dois grupos com objetivos distintos. O primeiro grupo engloba técnicas e ferramentas relacionadas à obtenção de informações estáticas a respeito do programa, quais sejam: fluxo de controle, referências cruzadas dos identificadores utilizados no programa, diagrama do fluxo de dados a partir do código fonte, hierarquia de chamadas das sub-rotinas, etc. [KEUF-90].

O segundo grupo é constituído por técnicas e ferramentas relacionadas à atividade do reconhecimento do projeto ("design"). Este reconhecimento "recria as abstrações de projeto a partir das combinações entre o código fonte, a documentação disponível, a experiência profissional e os conhecimentos gerais acerca do domínio de aplicação", visando reproduzir "as informações necessárias para que o programador entenda totalmente o software em termos *do que ele faz, como ele faz e por que ele faz*" [BIGG-89].

O objetivo deste segundo grupo é bastante abrangente, envolvendo a determinação da funcionalidade do programa, bem como o reconhecimento das decisões tomadas à nível de projeto do software. Vale destacar que, apesar da existência de pesquisas sobre as técnicas e ferramentas para

reconhecimento do projeto[2], as mesmas não se encontram ainda comercialmente desenvolvidas.

Portanto, a engenharia reversa envolve desde simples informações estáticas (por exemplo, determinação do fluxo de controle) até o total controle intelectual sobre o software em estudo.

1.2 Definição do Problema

Derivar a descrição funcional (Abstração) de um código fonte de um programa, visando facilitar a determinação *do que o programa faz* e *do como ele faz* uma determinada tarefa.

1.3 Objetivos da Pesquisa

Este trabalho de pesquisa propõe-se a desenvolver e implementar um modelo para abstração funcional, cujo objetivo é a determinação precisa do efeito de um programa, ou parte dele, sobre as suas variáveis em todas as situações encontradas no transcorrer do seu fluxo de controle. Vale observar que este modelo se aproxima dos objetivos do segundo grupo de modelos de engenharia reversa discutidos anteriormente.

O modelo proposto é baseado no trabalho de Hausler [HAUS-90] e tem como requisito básico a estruturação do código fonte, a qual permite que o programa seja constituído por várias partes identificáveis, passíveis de serem entendidas isoladamente com o objetivo de se conseguir o entendimento do todo.

O estudo do modelo proposto envolve a investigação das propriedades da estruturação que tornam possível a abstração, assim como dos mecanismos e técnicas que visam suportar a definição e construção de uma Ferramenta de Abstração Funcional (FAF), como as técnicas de "Trace-Table", "Trace-Table"

---

[2] Vide o modelo de engenharia reversa proposto por Biggerstaff [BIGG-89].

condicional, execução simbólica, "Slicer" de programas e análise das relações de recorrência em iterações.

1.4 Organização da Dissertação

Cap. 1 - Introdução

Este capítulo apresenta o problema a ser investigado dentro do contexto em que ele se insere.

Cap. 2 - Fundamentos da Abstração Funcional

Este capítulo introduz as estratégias, os conceitos básicos e os requisitos envolvidos no processo de entendimento de um programa. Ademais, analisa as técnicas de execução simbólica e "Slicer" de programas.

Cap. 3 - Modelo para Abstração Funcional

Este capítulo apresenta o modelo de abstração funcional fundamentado nas propriedades matemáticas dos programas estruturados, o qual objetiva extrair o efeito do programa sobre as suas variáveis. Este modelo é dividido em dois processos: processo de análise, que consiste na segmentação do programa de acordo com as suas variáveis relevantes ao entendimento, e processo de síntese de cada segmento encontrado no processo de análise.

Cap. 4 - Arquitetura e Implementação da Ferramenta de Abstração

Este capítulo apresenta a arquitetura, as estruturas e os algoritmos utilizados para implementar a Ferramenta de Abstração Funcional.

Cap.5 - Verificação Prática da Ferramenta de Abstração Funcional

Este capítulo demonstra que os resultados gerados pela Ferramenta de Abstração Funcional estão corretos e para um dado programa alvo, em adição, mostra a flexibilidade desta ferramenta em duas situações diferentes.

Cap. 6 - Conclusões e Trabalhos Futuros

Este capítulo apresenta as conclusões e sugestões de trabalhos futuros.

Apêndices

O apêndice A apresenta a especificação em YACC para a sintaxe da linguagem alvo deste trabalho. O apêndice B fornece a especificação formal do tipo abstrato árvore "And-Or", que tem como objetivo a representação do fluxo de dados e controle do programa. O apêndice C consiste em uma série de algoritmos que são utilizados no processo de padronização e análise do código fonte. O apêndice D apresenta uma série de funções que poderão ser utilizadas pelo usuário para sintetizar o efeito de iterações. O apêndice E ilustra algumas informações intermediárias geradas pela Ferramenta de Abstração Funcional implementada. O apêndice F apresenta uma análise completa e detalhada de um programa. Finalmente, o apêndice G retrata as principais interfaces da Ferramenta de Abstração Funcional.

Glossário

Segue-se aos apêndices um glossário com a definição de alguns termos utilizados neste dissertação.

Capítulo 2
Fundamentos da Abstração Funcional

Um dos maiores problemas na área de engenharia de software é a manutenção, o que já foi mencionado no capítulo anterior. Isto se deve, em parte, ao fato de que sem um adequado entendimento de um programa (seja ele um software básico ou de aplicação) é impossível mantê-lo, testá-lo ou validá-lo efetivamente, o que se reflete em sua qualidade. Pode-se afirmar que a qualidade de um software está diretamente relacionada às ferramentas disponíveis. Por sua vez, boas ferramentas para entendimento e validação dependem de bons mecanismos de análise de programas.

Neste capítulo são introduzidos os conceitos básicos existentes no processo de entendimento de programas, visando à formulação de um modelo de abstração funcional no Capítulo 3. Este modelo é totalmente baseado nos elementos existentes no próprio código fonte do programa e possui, como requisito básico, a estruturação deste código fonte.

2.1 Conceito de Abstração Funcional

A descrição das estratégias normalmente adotadas por um programador para o entendimento de um programa através de abstrações pode preceder a modelagem e construção de uma ferramenta que auxilie na obtenção da sua funcionalidade.

A abstração é o ato de extrair as informações essenciais e fundamentais para se realizar um particular propósito, desprezando-se as informações não relevantes [IEEE-83].

Nessa pesquisa, a ação de abstrair é direcionada para o entendimento do código fonte de um programa com o objetivo de determinar *o que um programa faz* e *como ele faz* uma determinada tarefa. O autor da pesquisa assume que é possível construir ferramentas, tendo por base o processo cognitivo através do qual os programadores tentam derivar a funcionalidade de um programa. Neste caso, existem três tipos de estratégias diferentes relacionadas ao processo cognitivo em questão [SHNE-79,CORB-89]:

i) *Estratégia "Bottom-Up"*

Esta estratégia prevê que o programador leia o código fonte sistematicamente com o objetivo de adquirir um gradual e completo entendimento do programa. Neste processo, vários itens similares ou relacionados são agregados conceitualmente para que formem uma única abstração [CURT-85]. Como exemplo, pode-se considerar a seguinte fatia de um código fonte como uma unidade do programa a ser abstraída:

```
sum = 0;
for(i=0; i < n; i++) {
     sum = sum + x[i];
}
```

Neste exemplo, estas quatro linhas de código podem ser sintetizadas em "calcular a soma dos elementos de um vetor x". O programador poderá, então, trabalhar com esta síntese no restante do programa.

Esta estratégia produzirá resultados melhores se o programador conseguir determinar abstrações concisas para programas ou suas partes mais complexas. Todavia, ela é mais indicada quando se tem pouco conhecimento sobre o código fonte a ser entendido.

ii) *Estratégia "Top-Down"*

Esta estratégia permite que o programador utilize a sua experiência no sentido de confirmar as suas expectativas a respeito de como o programa foi construído [CORB-89]. Por exemplo, se o programador está trabalhando com um programa de folha de pagamento, ele pode supor que certas construções foram utilizadas no código, isto é, que exista um arquivo de empregados cujos campos são: nome, matrícula, cargo, etc. Além disso, ele pode pressupor a existência de um processo para inserir ou eliminar funcionários, de rotinas de impressão e de tratamento de erros e exceções, etc.

O programador lista as suas expectativas e as localiza no código, verificando o que está sendo executado com o objetivo de ganhar um melhor entendimento sobre o programa. Entretanto, o fato de que alguma das expectativas não seja plenamente atendida permitirá que o programador ganhe mais experiência em relação ao domínio de aplicação do programa [CORB-89].

iii) *Estratégia Concomitante*

Esta estratégia utiliza-se das outras estratégias "Bottom-Up" e "Top-Down" para permitir o entendimento de um programa, ou seja, considera a sistematização do processo de análise da estratégia "Bottom-Up" associada à experiência passada do programador [CORB-89].

A escolha de uma das estratégias apresentadas depende da familiaridade do programador com a implementação e de sua experiência. Pode-se afirmar, também, que a utilização da estratégia "Bottom-Up" busca entender o programa por um processo indutivo, produzindo resultados mais concretos e

relacionados ao funcionamento do programa. Por outro lado, a estratégia "Top-Down" é um processo dedutivo de entendimento, cujos resultados são mais gerais e relacionados com *o que o programa faz*. A Figura 2.1 ilustra a diferença existente entre as duas primeiras estratégias.

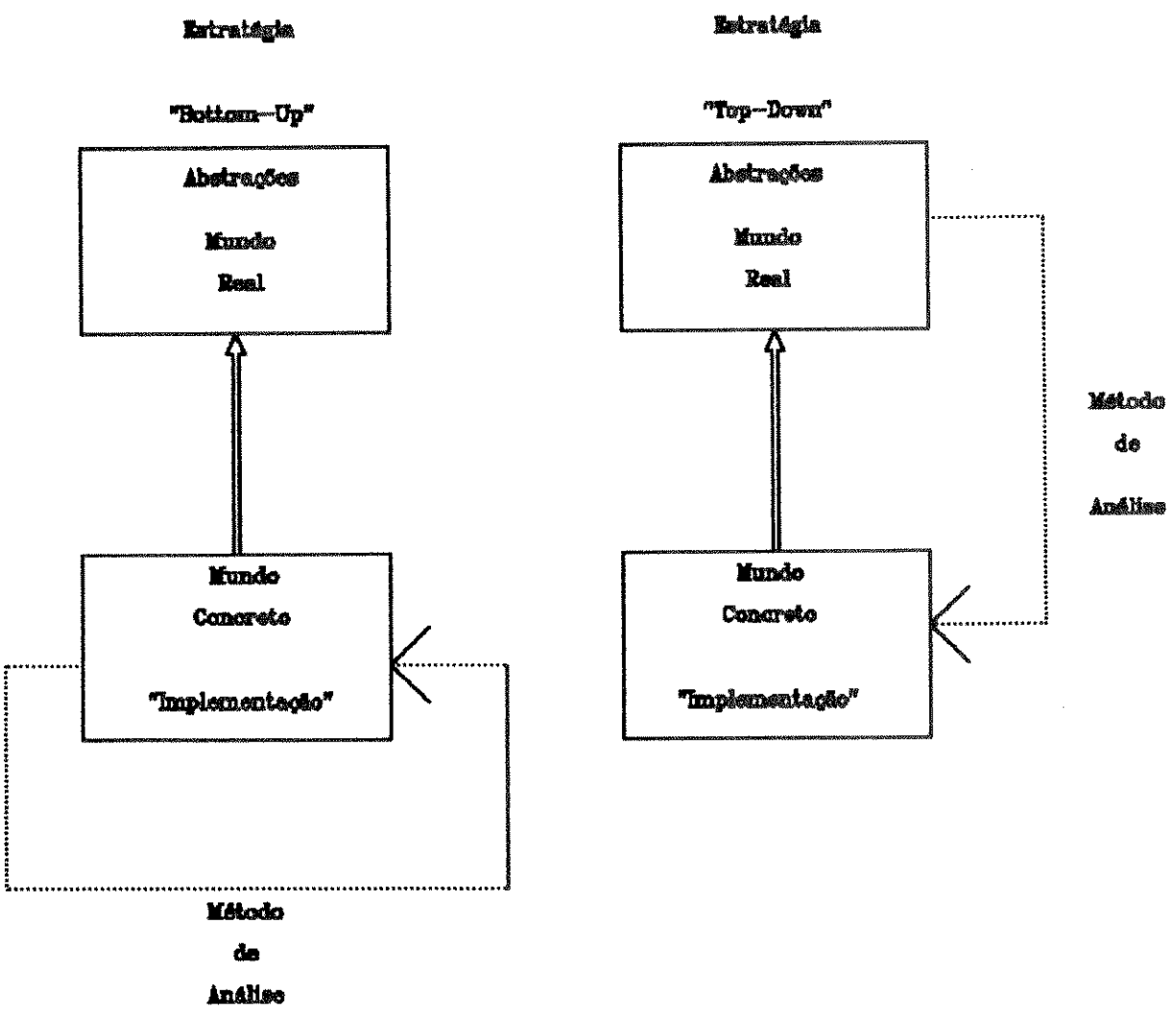


Figura 2.1: Diferença entre as Estratégias "Bottom-Up" e "Top-Down".

A importância dessas estratégias relacionadas ao processo de cognição reside no fato de que elas podem ser utilizadas em modelos e ferramentas para abstração de programas[1]. Alguns desses modelos são listados a seguir, juntamente com as suas vantagens e desvantagens:

i) Modelos de Entendimento Fundamentados na Estratégia "Bottom-Up"

Estes modelos determinam as unidades de abstração (partes de um programa) e procedem à substituição destas unidades por suas abstrações, visando determinar a funcionalidade do programa. Como exemplo, pode-se citar o modelo de entendimento de programas proposto por Basili [BASI-82] e o modelo de abstração funcional de Hausler et al [HAUS-90].

Vantagens

Algumas vantagens dos modelos fundamentados na estratégia "Bottom-Up" são listados a seguir:

- Produzem resultados mais concretos, permitindo que estes possam ser utilizados para validação, teste e auxílio ao entendimento do código fonte;
- Não dependem da variação sintática da implementação;
- O programa poderá ser todo construído com identificadores, cujos nomes não guardem qualquer relação com o seu objetivo (por exemplo, um programador poderá rotular uma variável com o nome "xx", escondendo, desta forma, o seu significado); e
- Apesar da inexistência de ferramentas disponíveis no mercado, esta estratégia fornece uma forte base para que se produza uma ferramenta automatizada.

---

[1] O autor define ferramenta de abstração como Software Básico utilizado para derivar uma abstração em mais alto nível de um programa, visando facilitar o seu entendimento.

Desvantagens

Algumas desvantagens dos modelos fundamentados na estratégia "Bottom-UP" são listados a seguir:

- Não produzem resultados muito abstratos, necessitando de análises posteriores para determinar *o que o programa faz*; e
- Os resultados obtidos podem ser muito complexos.

ii) Modelos de Entendimento Fundamentados na Estratégia "Top-Down"

Estes modelos tentam agregar as informações informais existentes em um programa (comentários, nomes de identificadores, etc) ao domínio de aplicação do programa, objetivando determinar *o que o programa faz*. O trabalho desenvolvido por Rich [RICH-90] é um bom exemplo de uma proposta de um modelo de entendimento fundamentado na estratégia "Top-Down". Este modelo tenta derivar a funcionalidade do programa por meio de uma base de conhecimentos sobre clichês comumente utilizados pelos programadores.

Vantagens

Algumas vantagens dos modelos fundamentados na estratégia "Top-Down" são listados a seguir:

- Produzem resultados bem abstratos; e
- Os resultados obtidos são mais fáceis de serem interpretados.

Desvantagens

Algumas desvantagens dos modelos fundamentados na estratégia "Top-Down" são listadas a seguir, valendo observar que tais desvantagens foram identificadas por Rich [RICH-90]:

- Dependem muito da variação sintática da implementação;
- Não são fortemente automatizáveis; e
- Nem todo programa é construído inteiramente com "clichês", inviabilizando o reconhecimento de padrões.

Tendo em vista as sérias limitações para automatização dos modelos baseados nas estratégias "Top-Down", o autor optou por basear sua pesquisa na determinação de um modelo para abstração funcional de programas fundamentado em uma estratégia "Bottom-Up".

2.2 Fundamentos Básicos

Nesta seção são introduzidos os fundamentos básicos de um modelo para abstração funcional baseado em uma estratégia "Bottom-Up". O modelo de abstração aqui proposto baseia-se no trabalho de Hausler et al [HAUS-90] e tem como suporte teórico a aplicação dos conceitos da teoria funcional [LING-79, ZELK-90, MILL-75, BASI-75], combinada com técnicas de análise do fluxo de programas, com a aplicação da técnica de "Slicer" de programas [WEIS-82], com as simplificações algébricas [BUCH-82], com a execução simbólica [CHEA-79, KING-76] e, por fim, com os construtores e conceitos da programação estruturada [LING-79].

2.2.1 Estruturação do Programa

O primeiro passo para aumentar o entendimento sobre um código fonte é a sua estruturação e padronização, pois "quando não existem desvios arbitrários no fluxo de controle, pode-se entender e documentar um programa através de sistemáticas leituras do código". Desse processo de leitura sistemática, "deve resultar um total controle intelectual sobre as ações do programa, permitindo, assim, executar modificações e evoluções" [HAUS-90].

Um programa estruturado tem apenas um ponto de entrada e um ponto de saída e seu diagrama de fluxo é composto por "nós-funções"(2), os quais

(2) Os "nós-funções" são nós de um diagrama de fluxo que possuem apenas um ponto de entrada e um ponto de saída.

representam duas outras classes de programas: programa próprio e programa primo.

Programa Próprio

Um programa próprio possui um diagrama de fluxo com apenas um ponto de entrada e um ponto de saída e, ainda, existe um caminho a partir da entrada até a saída para todo e qualquer nó considerado [LING-79]. A Figura 2.2 ilustra alguns exemplos de programas próprios e não próprios.

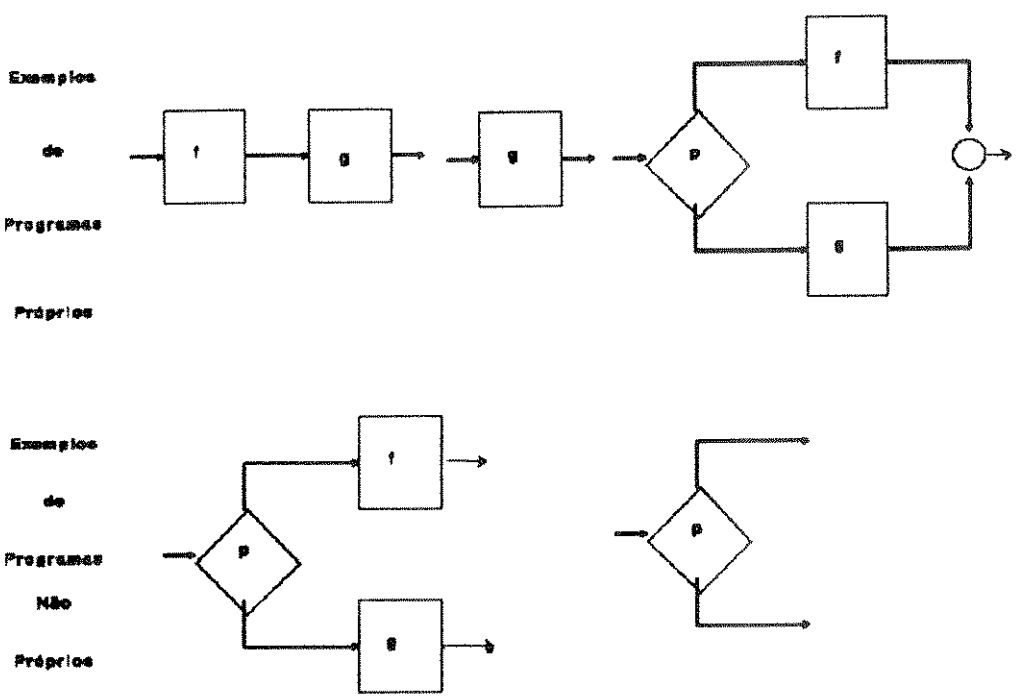


Figura 2.2: Programas Próprios e não Próprios

Programa Primo

Programa primo é todo programa próprio que não possua nenhum outro subprograma próprio, exceto ele mesmo. A importância dos programas primos se encontra explicitada na idéia de que um programa estruturado é um programa construído a partir da composição de programas primos [LING-79]. A Figura 2.3 mostra alguns exemplos de programas primos e não primos.

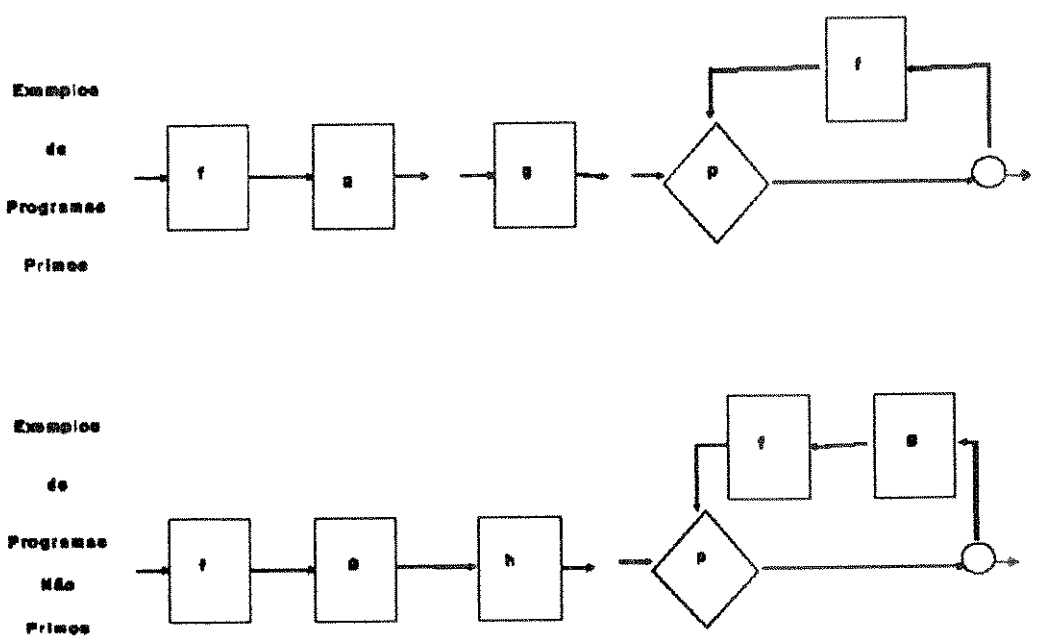


Figura 2.3: Programas Primos e não Primos

Cabe observar que qualquer programa construído a partir da junção de vários programas próprios pode ser decomposto em uma hierarquia de primos, na qual um primo de um determinado nível serve como "nó-função" do próximo nível da hierarquia, como pode ser visto na Figura 2.4 [LING-79].

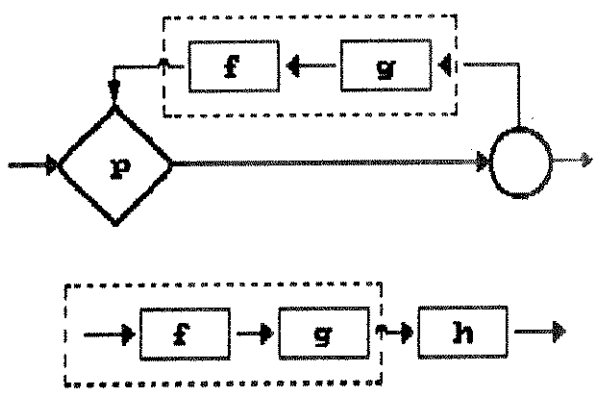


Figura 2.4: Decomposição em Programa Primos

Um programa primo pode ser de qualquer tamanho e complexidade. Todavia, existe um grupo de programas primos que é fundamental na estruturação de programas. Eles são conhecidos por pequenos primos e constituem um conjunto particular de construtores, que permitem ao programador manter o controle do programa em desenvolvimento [MILL-75, HAUS-90, BASI-75]. Este controle envolve a divisão do programa em peças menores de mais fácil compreensão.

Os construtores ou pequenos primos e as suas funções computadas se encontram abaixo relacionados, cabendo esclarecer que o esquema de notação é baseado nos axiomas(3) funcionais de Mills [MILL-75].

- **Significado dos Símbolos Utilizados nos Axiomas Funcionais**

P -> Denota um programa ou parte dele.

Y -> Denota o vetor de estado do programa, ou seja, $Y = (y_1, y_2, \ldots y_n)$, onde Y é uma lista de todas as variáveis e os valores de um programa P.

[P] -> Denota a função computada por P. (Assume-se que P termina)

[P](Y) = Y' -> Y'representa a nova lista de valores obtida após a execução de P sobre a lista de valores de Y.

---

(3) Proposições admitidas como verdadeiras, das quais pode-se deduzir novas proposições de uma teoria ou de um sistema lógico ou matemático.

[B](Y) -> Valor boleano (falso ou verdadeiro).

Si -> Lista ou bloco de comandos.

- <u>Axiomas Funcionais para os Construtores de um Programa</u>

Atribuição: P representa yk := f(Y) para 1 <= k <= n.

O axioma funcional associado a este construtor é:

$$[y_k = f(Y)]\ (Y) = (y_1, y_2, \ldots, f(Y), \ldots, y_n)$$

Isto é, a nova lista de valores $(y_1, y_2, \ldots, f(Y), \ldots, y_n)$ é praticamente a mesma de Y, com exceção da variável y_k, que possui um novo valor determinado por f(Y).

Decisão: a. P representa if B then S1 else S2 fi

O axioma funcional associado a este construtor é:

[P] = [if B then S1 else S2 fi] (Y)

$$= \begin{cases} \text{[S1](Y)} & \text{se [B](Y) é verdadeiro} \\ \text{[S2](Y)} & \text{se [B](Y) é falso} \end{cases}$$

Em outras palavras, se a condição B é verdadeira, então a função de P é a mesma de S1; caso contrário, a função de P é a mesma de S2.

b. P representa if B then S1 fi

O axioma funcional relacionado a este construtor é:

[P] = [if B then S1 fi](Y)

$$= \begin{cases} \text{[S1](Y)} & \text{se [B](Y) é verdadeiro} \\ \text{Y} & \text{se [B](Y) é falso} \end{cases}$$

Neste caso, se B é falso, a lista de variáveis Y permanecerá inalterada.

Composição: P representa S1;S2.

P é uma lista de comandos, formada pela lista de comandos S1 seguida da lista S2. O seu axioma funcional é:

$$[P] = [S1;S2](Y) = [S2]([S1](Y))$$

Ou seja, a nova lista de valores, após a execução de P, é igual à obtida pela execução de S2 sobre o resultado obtido por S1 com a lista de valores Y.

Iteração: While B do S1 od

Neste caso, S1 é uma lista de comandos que deverá ser executada enquanto B for verdadeiro. Caso B seja falso, S1 não será executado. O axioma funcional para este construtor é:

$$\begin{aligned} [P] &= [\text{while B do S1}]\,(Y) \\ &= [\text{if B then S1; while B do S1}]\,(Y) \\ &= [\text{if B then S1; P fi}]\,(Y) \\ &= \begin{cases} [P](S1(Y)) & \text{se } B(Y) \text{ é verdadeiro} \\ Y & \text{se } B(Y) \text{ é falso} \end{cases} \end{aligned}$$

Isto é, o programa P pode ser visto como uma função recursiva.

Chamada a Sub-Rotina ("Call"): P representa Call Sub(W)

P é uma chamada à sub-rotina ("Sub"), que tem como parâmetros a lista W, de tal modo que: Y = (W, Z). O axioma funcional para este construtor é:

$$[P](Y) = ([Sub](W), Z)$$

Isto é, a execução de P sobre Y é equivalente à execução de Sub sobre W sem, entretanto, alterar qualquer variável em Z.

Estes construtores permitem subdividir um programa estruturado em uma hierarquia formada de subprogramas estruturados, que podem ser entendidos independentemente do resto do programa [BASI-75]. Ou seja, dado qualquer programa escrito a partir dos construtores fornecidos anteriormente, pode-se criar esta hierarquia, atribuindo-se a cada construtor um nível

hierárquico particular, sendo que o nível mais baixo da hierarquia corresponde aos comandos de atribuição. Considere o seguinte fragmento de programa como exemplo:

```
1. z = 0;
2. y = b;
3. while(y > 0) {
4.    y = y - 1;
5.    z = z + a;
6. }
```

A Figura 2.5 ilustra a hierarquia gerada a partir deste programa.

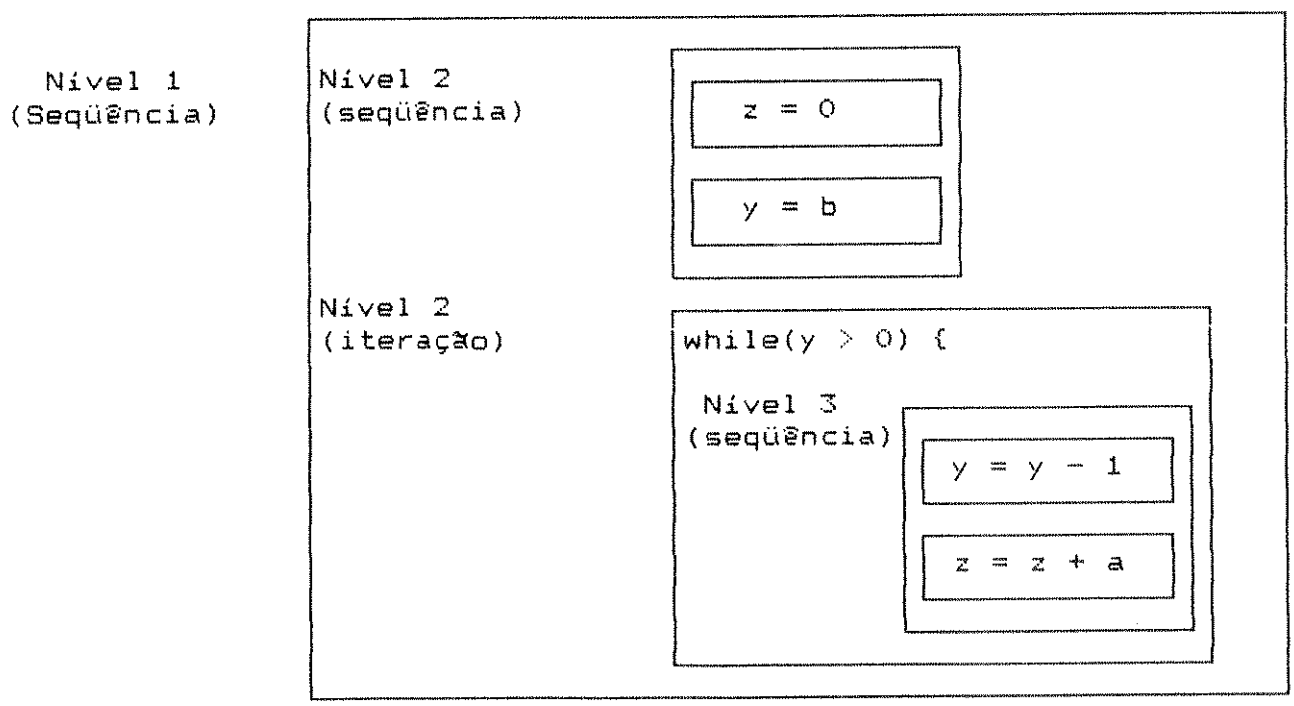


Figura 2.5: Hierarquia de Programas Primos

Esta hierarquia, conhecida como decomposição em programas primos, explica a propriedade dos programas estruturados que torna possível a abstração, já que um programa estruturado é formado por expansão funcional de primos e a obtenção de sua funcionalidade pode ser reduzida à determinação da função de cada categoria de primo [HAUS-90]. Cabe observar que o programa primo é a unidade de abstração básica no processo de entendimento "Bottom-Up" adotado.

Cada abstração representa uma função no sentido matemático estrito, isto é, fornecido um vetor de entrada E e um vetor de saída S, a função do primo define regras para achar S a partir de E (S = f(E)) [BASI-75]. Sendo assim, pode-se derivar a funcionalidade de um programa completo a partir da composição das funções dos vários primos, dado que um programa primo pode ser intercambiado por sua função, como indica o axioma de substituição de Mills [LING-79].

Resumidamente, o axioma explicita que programas estruturados são construídos a partir de programas estruturados menores, os quais podem ser utilizados para sumarizar os maiores. Este axioma é utilizado na derivação dos princípios para leitura, validação e escrita de programas. Assim, a Figura 2.6, extraída de Linger [LING-79], ilustra a escrita de um programa (expansão funcional), a leitura (abstração funcional) e a validação (comparação das funções conhecidas e de suas expansões). Estas expansões e abstrações são operações algébricas em programas estruturados [LING-79, HAUS-90, MILL-75, BASI-75].

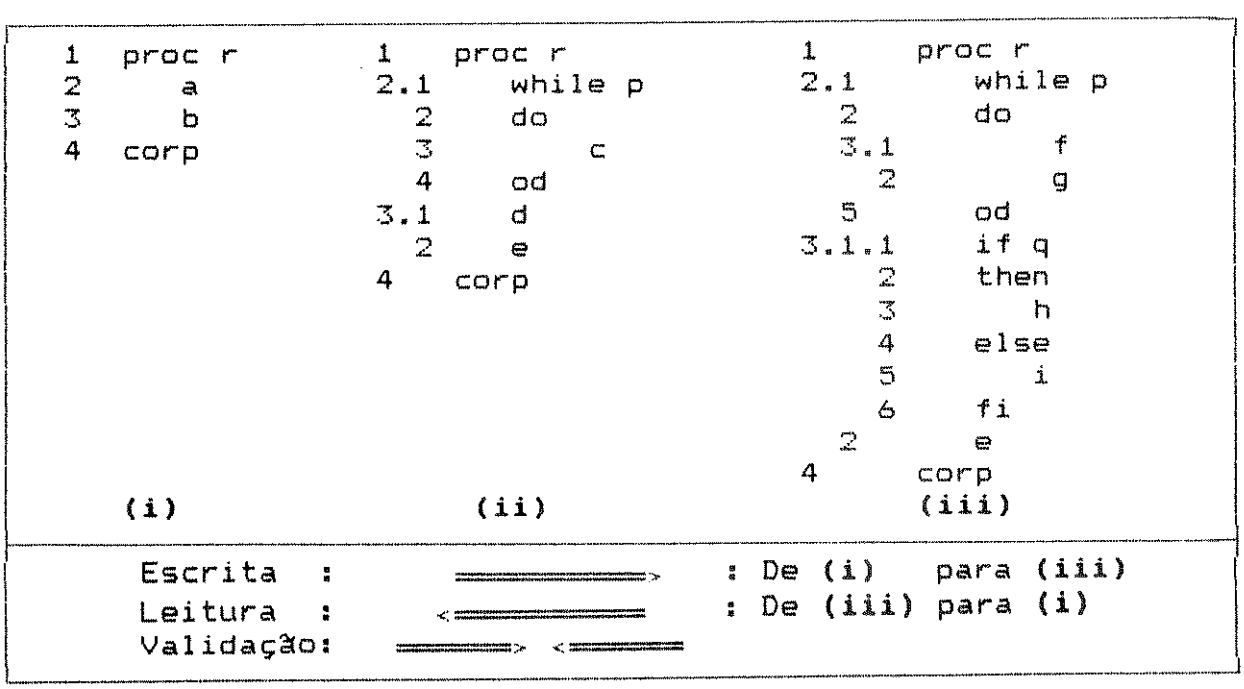


Figura 2.6: Leitura, Escrita e Validação de Programas Estruturados [LING-79].

Os programas primos serão analisados e as suas funções serão derivadas para representar o seu comportamento durante o processo contínuo de abstração. Uma função-programa de um primo P, denotada [P], é definida como sendo todas as possíveis execuções de P a partir de uma entrada até sua saída [BASI-82]. Neste contexto, a definição do conceito de execução simbólica torna-se fundamental para a geração dessas possíveis execuções.

2.2.2 Execução Simbólica

A execução simbólica é uma técnica de verificação e análise em que a execução do programa é simulada. Nesta simulação, utilizam-se símbolos no lugar dos dados de entrada e as saídas geradas pelo programa são expressas em termos de expressões lógicas ou matemáticas [IEEE-83].

Apesar de nenhuma ferramenta comercial ter sido ainda produzida [CLAR-85], três técnicas básicas foram desenvolvidas nos anos 70: avaliação simbólica para um determinado caminho do programa, avaliação simbólica dinâmica e execução simbólica global. Estas técnicas são descritas a seguir:

- A avaliação simbólica dependente de caminhos de execução gera uma descrição simbólica do programa para um determinado conjunto de caminhos fornecidos externamente. Esta descrição pode ser obtida, empregando-se duas técnicas: expansão direta ou substituição inversa.

  A técnica de expansão direta parte do primeiro comando do programa e gera expressões simbólicas e lógicas para cada comando existente no caminho requerido. Os trabalhos de King [KING-76] e Clarke [CLAR-76] são exemplos dessa técnica.

  A técnica de substituição inversa começa no ponto de saída do programa e termina no seu ponto de entrada. Neste processo, cada comando é avaliado para que sejam desenvolvidas expressões simbólicas das variáveis. Essa técnica foi desenvolvida para

derivar as expressões lógicas que determinam o domínio do caminho desejado. O trabalho de Huang [HUAN-75] é um exemplo desta técnica.

- A avaliação simbólica dinâmica gera uma representação do programa para um determinado conjunto de dados de entrada. Esta representação permite que se mantenha, além dos valores simbólicos de todas as variáveis, o resultado obtido pela aplicação dos valores fornecidos na entrada para cada expressão simbólica gerada. Cabe observar que os valores de entrada sensibilizam um determinado caminho do programa. Veja Fairley [FAIR-75], para maiores detalhes.

- A avaliação simbólica global gera uma representação simbólica para as variáveis e domínios de todos os caminhos do programa. Neste sentido, as iterações são tratadas separadamente com o objetivo de se obter funções que capturem o seu efeito em relação a cada variável definida em seu corpo. Ademais, estas funções englobam uma classe de caminhos, sendo que cada caminho difere do outro apenas pelo número de ciclos da iteração. Este método produz bons resultados para documentação, entendimento e teste de programas. O único trabalho encontrado pelo autor desta pesquisa sobre avaliação simbólica global foi o de Cheatham et al [CHEA-79].

A execução simbólica global aproxima-se mais dos objetivos desta pesquisa de mestrado, mantendo, inclusive, uma relação próxima à proposta de Hausler et al [HAUS-90]. A principal diferença entre essas duas propostas é a forma com que um programa é percorrido para se obter a descrição funcional, pois, enquanto a execução simbólica global atravessa o programa a partir de seu ponto de entrada até o seu ponto de saída, visando obter as expressões simbólicas e lógicas, a proposta de Hausler percorre o programa primo a primo, possibilitando que o resultado final da abstração seja mais conciso. Apesar dessa diferença fundamental, aspectos importantes dessas duas propostas são fundidos no Capítulo 3, que detalha os pontos aproveitados de cada uma delas.

2.3 Modelo Geral para a Abstração Funcional

Os conceitos existentes na avaliação simbólica global são agregados à estratégia de avaliação primo a primo com o intuito de derivar um modelo geral para a abstração funcional proposto neste trabalho de pesquisa.

Seja P um programa estruturado formado por pequenos primos. P pode ser visto como uma função que mapeia elementos do domínio X para o contradomínio Z. Um elemento de X é um vetor de dados de entrada, x=(x1, x2, ..., xm), que corresponde a um único ponto no espaço m-dimensional dos possíveis valores de entrada. Da mesma maneira, um elemento de Z corresponde a um único resultado das variáveis de saída do programa, z=(z1, z2, ..., zn), no espaço dos possíveis resultados do programa. Todas as variáveis de P, sejam de saída ou intermediárias, são representadas no vetor y=(y1, y2, ..., yw).

Considere, ainda, uma lista h, h=(h1, ..., hq), cujos elementos sejam pequenos primos ordenados de tal forma que a sua hierarquia em P seja obedecida. Neste caso, h1 representa um dos primos de mais baixo nível hierárquico.

Uma análise completa do programa para um determinado elemento de X se processa a partir da seguinte composição funcional:

$$y = hq(hq-1(\ldots h2(h1(x))) \ldots))$$ [4]

Cada programa primo (hi) será abstraído de acordo com o seu tipo (vide Capítulo 3), obedecendo à seguinte especificação:

---

[4] A semântica de cada tipo de pequeno primo foi descrita anteriormente através dos axiomas funcionais de Mills.

i) O primo "hi" é Uma Seqüência de Atribuições

Neste caso, o efeito da seqüência será especificado a partir de atribuições concorrentes, cuja sintaxe é:

y1, y2, y3, ..., yn = expr1, expr2, ..., exprn

Este comando significa que as expressões expr1, expr2, ..., exprn são avaliadas independentemente e, somente então, seus valores são atribuídos a y1, y2, ..., yn, respectivamente.

ii) O primo "hi" é Uma Seqüência que Possui Atribuições Condicionais

Neste caso, o efeito desta seqüência será especificado a partir de atribuições condicionais concorrentes, cuja sintaxe é:

(p1 -> A1 ¦ p2 -> A2 ¦ .... ¦ pr -> Ar), onde:

p1, p2, ..., pr são predicados e A1, A2, ..., An são atribuições simples ou concorrentes.

Estas atribuições indicam que uma particular atribuição (Ai) será ativada quando algum pi for verdadeiro, se nenhum pi for verdadeiro, o comando será indefinido. Observa-se que as atribuições concorrentes condicionais são geradas pelos comandos de decisão do programa.

iii) O Primo "hi" é Uma Decisão

Neste caso, as regras condicionais do construtor para decisão devem ser transformadas em regras disjuntas, visando compatibilizar a abstração do primo em questão com o axioma funcional para este construtor. Por exemplo:

[if P then G else H] = (p -> [G] | ~p -> [H])

As funções G e H podem, em casos mais gerais, representar regras condicionais, o que ressalta a importância da utilização das regras disjuntas. Considere o exemplo abaixo que não está na forma disjunta:

(p1 -> (q11 -> r11 ¦ q12 -> r12) ¦ p2 -> (q21 -> r21 ¦ q22 -> r22))(1)

Neste exemplo, "p" e "q" representam predicados e "r", ações. Se os predicados "p1" e "p2" fossem distribuídos nas regras internas, seria obtido:

(p1^q11 -> r11 ¦ p1^q12 -> r12 ¦ p2^q21 -> r21 ¦ p2^q22 -> r22) (2)

Esta distribuição parece estar correta, mas ela não é válida para o seguinte caso:

- "p1" e "p2" sejam verdadeiros,
- "q11" e "q12" sejam falsos e
- "q21" ou "q22" seja verdadeiro.

pois a regra original (1) é indefinida, o que não ocorre com a regra gerada pela distribuição dos predicados (2).

A única maneira de se proceder à distribuição dos predicados "p1" e "p2" é através do uso de regras disjuntas, isto é,

(p1^q11->r11 ¦ p1^q12->r12 ¦ ~p1^p2^q21->r21 ¦ ~p1^p2^q22->r22)

tornando a distribuição válida.

iv) O Primo é Uma Iteração:

A função programa derivada para seqüência e decisão é diretamente utilizável no processo de composição. Porém, isto não é válido para iteração, porque ela pode gerar um número muito grande, às vezes infinito, de possíveis caminhos de execução do programa. Análises adicionais são necessárias para solucionar este problema, visando representar a iteração como uma expressão fechada que descreva o seu efeito [MUCH-81].

A Seção 2.4 descreve detalhadamente o construtor para iteração, porém, a técnica de abstração funcional deste construtor é postergada para o Capítulo 3. Por ora, assume-se que qualquer iteração possa ser representada como uma função recursiva.

Durante este processo de análise e composição funcional, os valores de todas as variáveis são mantidos como expressões algébricas em termos de seus nomes simbólicos. Em qualquer ponto desta composição, os valores parciais e o domínio das variáveis do programa são denotados por VP[h1i, R, j], onde h1i indica a composição de $h1 \circ h2 \circ \ldots \circ hi$, i <= q, R representa o conjunto de possíveis casos de execução do programa até um determinado ponto da análise do mesmo e j indica o número total de casos existentes em R. Um elemento de R é uma dupla ordenada, (CE[i], Ai), $i \leq j$, na qual o primeiro membro, CE[i], corresponde ao domínio das expressões simbólicas existentes na atribuição concorrente Ai.

No final, a computação executada pelo programa é denotada apenas pelas variáveis de VP[h1q, R, r], que façam parte do conjunto de variáveis de saída do programa (r pode ser traduzido como o número de caminhos do programa, se a restrição em relação à iteração for levada em consideração), produzindo o seguinte resultado:

```
Caso: CE[1]: A1 : z1 = C11
                  z2 = C12
                     .
                     .
                     .
                  zn = C1n
        .
        .
        .
      CE[r]: Ar : z1 = Cr1
                  z2 = Cr2
                     .
                     .
                     .
                  zn = Crn
Fim.
```

Os símbolos acima possuem os seguintes significados:

CE[i] -> Indica uma determinada condição de execução, i ≤ r.

Cik -> Representa as funções computadas pelo programa para o caso "i", k ≤ n.

n -> Representa o número de variáveis de saída do programa.

r -> Representa o número total de casos executados pelo programa.

O modelo geral para abstração funcional e a semântica do "Caso" (resultado acima) são utilizadas nos Capítulos 3 e 4 para descrever um modelo de abstração funcional.

2.4 Conceitos Básicos para o Construtor de Iteração

A iteração pode ser caracterizada como um programa que é executado um número finito de vezes até que uma condição de parada seja satisfeita. Sendo assim, ela pode ser generalizada da seguinte forma:

WHILE B do S

onde B é uma expressão boleana e S é um bloco de comandos.

Observa-se que o programa só termina se, e somente se, o bloco S influenciar B de tal forma que seja possível à expressão B se tornar falsa após um número finito de execuções. Neste caso, deve ocorrer pelo menos uma atribuição dentro de S das variáveis que se encontram em B para que a condição de finalização possa ser satisfeita [WIRT-78].

Partindo-se da notação utilizada por [WIRT-78], pode-se escrever um programa "loop"(5) na forma geral representada na Figura 2.7.

```
V := v0
While p(V) do V := f(V)
```

Figura 2.7: Forma Geral do "Loop"

Nesta forma geral, o conjunto **V** denota todas as variáveis do "loop", **p** denota um predicado e **f**, uma função. O conjunto **V** assumirá a seqüência de valores **v1, v2, v3, ..., vn**. Neste contexto, considere **vi** o valor de **V** antes da i-ésima execução de **S**, tal que **vi** obedeça às seguintes propriedades da Tabela 2.1.

---

(5) Os termos iteração e "loop" são intercambiáveis neste trabalho de pesquisa.

1. vi = f(vi-1)	vi é definida em função de seu valor no ciclo anterior, para todo i > 0.
2. $_{1}$p(vn)	Esta propriedade representa a condição de parada. Se o n-ésimo valor de V for aplicado à condição do "loop", produzirá um resultado falso.
3. p(vi)	Esta propriedade representa a condição para um novo ciclo do "loop". Se o i-ésimo valor de V (i < n) for aplicado à condição do "loop", produzirá um resultado verdadeiro.

Tabela 2.1: Propriedades das Variáveis do "Loop" para um Determinado Ciclo [WIRT-78].

O "loop" usualmente implementa uma "fórmula" computacional recorrente. Desta forma, um caminho que pode ser escolhido para se obter abstrações do "loop" é a direção de esforços no sentido de desenvolver as relações de recorrência nas situações possíveis. Entretanto, o "loop" pode ser dividido em segmentos independentes para que seja viável o desenvolvimento de uma técnica de abstração que é descrita na Seção 3.3.3.

2.4.1 Elementos Computacionais Básicos da Iteração

Segundo Waters [WATE-79], o "loop" possui três elementos computacionais básicos que formam a sua estrutura lógica, quais sejam: computações independentes, filtros e "loop" básico.

Computações Independentes

Um "loop" é formado por um conjunto de computações e cada uma delas pode ser caracterizada como um subconjunto mínimo, formado por todos os

comandos e variáveis que não afetam quaisquer outras computações do corpo do "loop" em questão [WATE-79].

Uma computação é geralmente constituída de duas partes: um corpo e uma iniciação. O corpo é um código com apenas um ponto de entrada e um ponto de saída, sendo localizado em alguma parte do fluxo de controle do "loop". A Figura 2.8 contém um exemplo de "loop" com duas computações independentes [WATE-79].

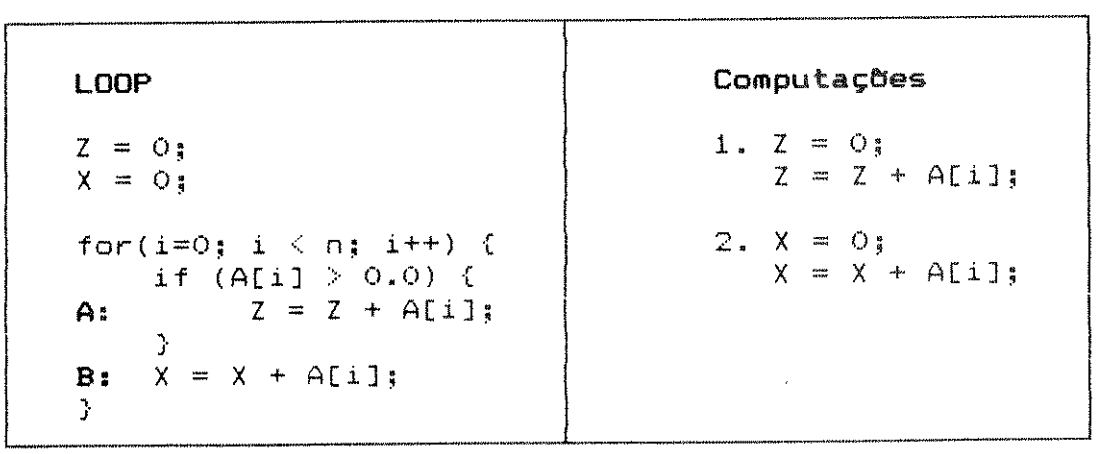


Figura 2.8: Exemplo de Computações Independentes [WATE-79]

A iniciação constitui-se de comandos localizados antes do "loop". Como pode ser observado na Figura 2.8, "Z=0" é uma iniciação responsável pela determinação de um valor que será utilizado no corpo do "loop".

Cada computação pode ser entendida isoladamente, sendo considerada, numa visão mais abstrata, como receptora de um conjunto de valores (um para cada variável referenciada) e produtora de uma seqüência de valores (uma para cada variável atribuída) [WATE-79].

Cabe apenas salientar que relações de recorrência podem ser desenvolvidas para especificar o comportamento de uma determinada computação. Considere o exemplo anterior, a computação "X=0; X = X + A[i]" produz a relação de recorrência "$X_0 = 0 \wedge X_i = X_{i-1} + A[i]$", que descreve

o seu comportamento. Observe que esta relação de recorrência computa a soma do vetor "A", ou seja,

$$X_{i+1} = \sum_{j=0}^{i} A[j].$$

Um experimento realizado por Waters revelou que 89% das computações dos "loops" estudados implementavam uma das seguintes formas recorrentes: soma, contador, máximo, mínimo ou uma relação de recorrência trivial, na qual o valor anterior da variável não aparece (por exemplo, "Xi = 2*Zi"). Tal experimento não foi muito abrangente, porém mostrou que a análise das iterações via resolução das relações de recorrência é uma estratégia razoável.

Filtro

O filtro pode ser considerado um caso especial de computação que restringe uma seqüência de valores de entrada. Ele pode ser composto com as computações independentes para determinar quais os valores que estarão disponíveis para o processamento em questão [WATE-79].

O exemplo da Figura 2.8 mostra que a posição de inserção do comando "Z = Z + A[i]" afetará o seu comportamento, ou seja, se ele for colocado na posição "A", somente os valores de "A[i]" maiores do que zero serão computados, e se for colocado na posição "B", todos os valores serão computados. Neste caso, a computação "Z=0; Z=Z+A[i]" terá a relação de recorrência "$Z_0=0 \wedge Z_i = Z_{i-1} + A[i]$". A existência de um filtro, que deixa passar apenas os valores positivos do vetor A, permite que a relação de recorrência seja melhor representada por:

$$Z_0=0 \wedge Z_i = Z_{i-1}+A[i] \wedge A[i] > 0$$

"Loop" Básico

Um resíduo permanece após a decomposição do "loop" em termos de suas computações e filtros [WATE-79]. Este resíduo corresponde ao que Pratt [PRAT-78] designou de controle computacional do "loop", isto é, tem uma iniciação, um teste e um comando que modifica a variável referenciada no teste, que correspondem ao "loop" básico [BISH-90].

A importância do "loop" básico reside no fato de que se pode tentar determinar uma expressão algébrica em função das condições de término, visando à determinação do número total de ciclos que o "loop" irá efetuar. Esta expressão é fundamental no processo de abstração funcional do "loop" que é detalhado no Capítulo 3.

Até este momento foi descrito um tipo de segmentação de programa dirigido apenas pelo seu fluxo de controle (decomposição em programas primos). Entretanto, com o objetivo de tornar possível a identificação e extração das várias computações executadas por uma iteração e permitir que o usuário da ferramenta possa analisar apenas algumas variáveis de seu interesse, faz-se necessário definir uma outra maneira de segmentação de um programa conhecida como técnica de "Slicer".

2.5 Técnica de "Slicer" de Programas

A técnica de "Slicer" foi proposta por Mark Weiser [WEIS-82, WEIS-84] em 1979 com o objetivo de dividir um programa de grande porte em pedaços menores para facilitar o seu entendimento e a sua manutenção.

O método de trabalho do "Slicer" é baseado na observação prática dos programadores, quando estes trabalham na manutenção de programas. Ou seja, durante o trabalho de "debugging", o programador se concentra num determinado subconjunto de interesse, procurando encontrar as partes do código fonte que afetam as variáveis que estão sendo estudadas [WEIS-82].

O "Slicer" é definido, de um modo formal, como uma técnica que seleciona os comandos e/ou variáveis de um programa, que afetam direta ou indiretamente um conjunto de variáveis a partir de um comando de interesse [SOBR-84]. Sendo assim, ele reduz o programa a uma forma mínima, que continua reproduzindo o comportamento do programa apenas para as variáveis desejadas. Considere a seguir alguns exemplos de "slices"(*) obtidos a partir do programa da Figura 2.9.

```
 1  main() {
 2      double x, y, z, total, sum;
 3
 4      scanf("%f %f", &x, &y);
 5      total = 0.0;
 6      sum   = 0.0;
 7      if (x == 1.0)
 8           sum = y;
 9      else {
10           scanf("%f", &z);
11           total = x*y;
12      }
13      printf("%f %f\n", total, sum);
14 }
```

Figura 2.9: Programa Original

A apresentação do "slice" para a análise da variável "z" é ilustrada na Figura 2.10. Cabe destacar que esta análise começou na linha 14, prosseguindo até a primeira linha do programa.

```
 1     main() {
 4          scanf("%f %f", &x, &y);
 7          if (x == 1)
 9          else
10               scanf("%f", &z);
14     }
```

Figura 2.10: "Slice" para a Variável "z"

(*) Conjunto de comandos selecionados pela aplicação da técnica de "Slicer".

A apresentação do "slice" para a variável "x" a partir da linha 14 e prosseguindo até a linha 1 se encontra na Figura 2.11.

```
 1     main() {
 4           scanf("%f %f", &x, &y);
14     }
```

Figura 2.11: "Slice" para a Variável "x"

A Figura 2.12 apresenta o "slice" da variável "total" a partir da linha 14.

```
 1  main() {
 4        scanf("%f %f", &x, &y);
 5        total = 0.0;
 7        if (x == 1.0)
 9        else {
11              total = x*y;
12        }
14  }
```

Figura 2.12: "Slice" para a Variável "total"

Um algoritmo foi proposto por Sobrinho [SOBR-84] para a aproximação do "Slicer", tendo em vista a ausência de solução para se encontrar um "slice" mínimo(7) de um programa. No algoritmo de Sobrinho, para se produzir um "Slicer" automático é requerido, em primeiro lugar, que o comportamento em análise seja expresso em termos dos valores de um conjunto de variáveis a partir de uma linha do programa. Esta especificação

(7) O "slice" mínimo é definido como a menor fatia de um código que captura o comportamento do programa para um dado critério.

é chamada de critério do "Slicer". Em segundo lugar, o algoritmo proposto exige que o programa seja representado na forma de uma árvore "And-Or", que representa o fluxo seqüencial de execução do programa. Esta árvore é um tipo abstrato de dados, que possui os seguintes tipos de nós:

OR - NODE : Nó que representa "IF's" ou "CASE's" das linguagens de programação, no qual apenas um dos filhos é executado, dependendo de alguma condição.

AND - NODE : Nó que representa um bloco de programa ("Begin", ...,"End"), no qual todos os filhos serão executados incondicionalmente e em seqüência.

LOOP - NODE: Nó que representa iterações ("FOR", "WHILE") das linguagens de programação, no qual todos os filhos serão executados de 0 a n vezes.

CALL - NODE: Nó que representa uma chamada à função ou procedimento.

NILL - NODE: Nó que representa um comando seqüencial simples como, por exemplo, comandos de atribuição. Este tipo de nó não possui filhos.

Assim, o algoritmo irá manipular esta árvore, percorrendo-a de baixo para cima, selecionando em cada nó as variáveis definidas e/ou referenciadas que afetam direta ou indiretamente o conjunto de variáveis de interesse.

2.5.1 Algoritmo da Técnica "Slicer"

Este algoritmo foi proposto por Sobrinho [SOBR-84] para aproximação de um "slice" de um programa. Vale salientar, entretanto, que este algoritmo não garante um "slice" mínimo.

Conceitos Básicos

É fundamental a definição de alguns conceitos básicos para que se possa entender o algoritmo:

1. **SC = <x, V, b>** é uma janela de "slicing", onde:
   - b: bloco de programa em que o "Slicer" será aplicado.
   - V: conjunto de variáveis de interesse.
   - x: comando, em b, onde o algoritmo irá começar.

2. **Slicing:** aplicação do algoritmo sobre um determinado programa.

3. **Slice:** conjunto de comandos selecionados por um slicing, os quais contribuem para o valor das variáveis contidas no conjunto V, exatamente antes do comando x ser executado.

4. **Para um comando y em b, sejam:**
   - Def [SC](y) = { v | v é uma variável recebendo valor em y};
   - Ref [SC](y) = { v | v é uma variável referenciada em y}; e
   - Con [SC](y) = { z | z é um comando que controla a execução de y}

5. **Current-Set [SC](y):** Este é um conceito chave no algoritmo, representando o conjunto que conterá o estado corrente e acumulado de todas as variáveis que afetam de alguma forma o critério, até o ponto do programa que está sendo analisado. Cabe salientar, contudo, que para cada nó da árvore, com exceção do nó "And", será gerado um "Current-Set" (CS) antes de se iniciar a próxima fase do algoritmo. Sendo assim, pode-se definir formalmente o CS:

   Current-Set [SC](y) = {v | se a ou b forem satisfeitos}

   a. <u>**se**</u> (x = y) <u>**então**</u>

   CS = V − (Def[SC](y) ∩ V) ∪ (Ref[SC](y) <u>se</u> Def[SC](y) ∈ V)

   Esta regra define o CS de partida do algoritmo, ou seja, o CS no ponto de partida é igual ao conjunto V das variáveis de interesse e, se a variável definida em y pertencer ao conjunto V, deve-se acrescentar à V as variáveis referenciadas em y e retirar a variável definida em y.

b. y é um predecessor imediato do comando z, tal que:

1. (v ∈ Current-Set [SC](z) e (v ∈ Def[SC](y))

A variável "v" é definida em y e pertence ao conjunto de variáveis que influenciam o comportamento do comando z. Por exemplo:

Comando y: **a** = k + m

Comando z: w = **a** + n -> No qual a pertence ao CS de z.

ou

2. (v ∈ Ref[SC](y)) e

2.1 (Def[SC](y) ∩ Current-Set[SC](z) ≠ 0)

A variável v é referenciada em y e a variável definida em y pertence ao conjunto das variáveis que controlam o comportamento do comando z. Por exemplo:

Comando y: **a** = **k** + 1

Comando z: w = **a** + 1 -> a pertence ao CS de z.

2.2 Existe um comando p predecessor imediato a q em b, tal que:

(Def[SC](p) ∩ Current-Set[SC](q) ≠ 0) e (y ∈ Con[SC](p)

Em outras palavras, y controla a execução do comando **p** e a variável definida em **p** pertence ao CS de **q**. Por exemplo:

comando y: if (**a** == 1) -> a referenciada em y

comando p: **w** = r + p -> y controla p

comando q: l = **w** + i -> w foi definida em p e pertence ao CS(q).

Logo, **a** deve entrar no CS(y).

6. Uma vez obtidos o CS para cada tipo de nó, com exceção do nó "And", deve-se aplicar os conceitos de SLICE() e Current-Set() em todos os tipos de nó da árvore "And-Or". Neste caso, define-se:

Cs_i = Current-Set (n_i, CS_{i-1}, b_i)

O "Current-Set" na i-ésima linha e no i-ésimo bloco é definido recursivamente em função do Current-Set anterior.

NILL-NODE:

se (Def(b) ∩ CS ≠ 0) ou

Existe z, sucessor à b, tal que ((b ∈ Con(z)) e (z ∈ SLICE) então {

- Current-Set(n, CS, b) = [CS − (CS ∩ Def(b))] U Ref(b)

  A preocupação é manter apenas as variáveis referenciadas no comando em questão, ou seja, retira-se a variável definida e inserem-se as variáveis referenciadas no comando. Por exemplo, se o comando "x = y + z;" afeta o critério, então, apenas as variáveis "y" e "z" devem entrar no CS.

- Slice(n, CS, b) = SLICE U {b}

}

Caso Contrário

- SLICE(n, CS, b) = SLICE
- Current-Set(n, CS, b) = CS

OR-NODE:

- $SLICE(n, CS, b) = SLICE(n_{then}, CS, b_{then}) \cup SLICE(n_{else}, CS, b_{else})$

  O "slice" deste tipo de nó é a união dos "slices" de cada filho, pois apenas um dos nós será executado e não se sabe a princípio qual deles.

- $Current\text{-}Set(n, CS, b) = Current\text{-}Set(n_{then}, CS, b_{then}) \cup Current\text{-}Set(n_{else}, CS, b_{else})$

  O "Current-Set" deste nó será equivalente à união dos "Current-Set" de cada filho.

AND-NODE:

- $SLICE(n, CS, b) = \bigcup_{i=1}^{m} SLICE(n_i, CS_i, b_i)$

  A função "slice" para o nó "And" é definida como a união dos "slices" de cada filho.

- $Current\text{-}Set(n, CS, b) = CS_m$

Basta considerar o CS do último filho para o Current-Set do nó "And", dado que ele possui a informação dos anteriores por recursividade.

LOOP-NODE:

- **SLICE(n, CS, b)** $= \bigcup_{i=1}^{m}$ **SLICE(n_i, CS_i, b_i)** $\bigcup_{i=1}^{m}$ **SLICE(n_i, CS_i, b_i)**

  O "Slice" de uma iteração é equivalente à união dos "slices" de cada filho, executados pelos menos duas vezes. O objetivo é garantir que todos os comandos, que possam afetar o critério, sejam considerados.

- **Current-Set(n, CS, b) = CS_m**

  Basta considerar o CS do último filho para o Current-Set do nó "Loop".

CALL-NODE:

- **Curren-Set(n, CS, b) = RAP(Current-Set(n, PAR(CS), b)), onde:**

  PAR(CS) = {v | [(v ∈ CS) e (v é global)] ou
  [(v é um parâmetro formal passado por referência ou resultado) e (Existe w | [(w ∈ CS) e (w é o parâmetro atual associado a v))]}

  RAP(PAR(CS)) = CS -> Recuperação do parâmetro formal.

- **SLICE(N, CS, b)**

  Este tipo de nó deve ser tratado como uma sub-árvore e, portanto, deve ser calculado o "slice" de toda a sub-árvore.

O algoritmo aceita um critério (conjunto de variáveis de interesse e uma linha de comando) e um bloco de comandos como entrada, para produzir um "slice" e um "current-set" baseados no critério. O resultado é um bloco de comandos que governa o comportamento e os valores do conjunto de variáveis de interesse. Tal algoritmo trabalha a partir do "current-set" de cada tipo de nó, para decidir se um comando deverá ou não ser incluído no resultado final, sendo cada "current-set" definido em termos do "current-set" anterior.

2.6 Conclusão

Este capítulo descreveu as estratégias envolvidas nos processos cognitivos de entendimento de um programa, isto é, as estratégias "Bottom-Up", "Top-Down" e concomitante, visando mostrar que é possível a utilização destas estratégias nos modelos de abstração funcional de programas.

Ademais, o capítulo mostrou que a estratégia "Bottom-Up" é mais adequada para se modelar uma ferramenta automatizada de abstração funcional. Neste sentido, as propriedades dos programas estruturados foram apresentadas para definição da unidade básica de abstração funcional (programa primo), a qual foi associada à execução simbólica global com o intuito de formular um modelo geral de abstração funcional. Este modelo pode ser resumido nos seguintes passos:

1. Obtenção da decomposição do programa em primos;
2. Análise de cada primo de acordo com o seu tipo, associando uma função a cada um deles; e
3. Composição funcional das abstrações parciais (primos) para obter gradativamente a funcionalidade do programa.

Por fim, procedeu-se à descrição mais detalhada do construtor para iteração no capítulo em questão, visando determinar os seus elementos computacionais básicos, e introduziu-se a técnica de "Slicer" com o objetivo de permitir que a iteração possa ser segmentada em termos de seus elementos computacionais e, em adição, que um programa possa ser abstraído apenas em função das variáveis de interesse.

Capítulo 3
Um Modelo para Abstração Funcional

Este capítulo descreve um modelo para abstração funcional de programas que consiste na adaptação da proposta feita por Hausler para uma ferramenta de abstração [HAUS-90]. O trabalho de Hausler, que proporciona o fundamento básico do processo de abstração (estruturação do programa e estratégia "bottom-up" para abstração), foi associado à técnica de execução simbólica global [CHEA-79] e ao método de análise de iterações [WATE-79], visando tornar o modelo proposto mais automatizável. Além do exposto, o modelo proposto pelo autor utiliza um novo conceito definido como Variáveis de Estado (Seção 3.2.1) com o objetivo de ganhar mais flexibilidade no processo de abstração funcional de programas.

Neste sentido, a Seção 3.1 apresenta diagramaticamente esse modelo conceitual, a Seção 3.2 descreve o processo de análise do programa e, por fim, a Seção 3.3 descreve as técnicas utilizadas para abstração dos programas primos.

3.1 Uma Síntese do Modelo para Abstração Funcional de Programas

Um programa real pode ser suficientemente complexo e inviabilizar a utilização prática de um modelo de abstração baseado nas informações contidas no seu código fonte. Com o objetivo de minimizar a complexidade intrínseca do programa, o modelo proposto o considera como sendo constituído de várias partes que são passíveis de identificação, extração e abstração.

Neste contexto, pode-se definir dois processos principais do modelo de abstração:

1. Processo de análise do programa, ou seja, segmentação e decomposição[1] do programa; e
2. Síntese ou abstração[2] de cada "parte" do programa encontrada no processo 1.

O processo de análise se dedica à investigação do código fonte com o objetivo de identificar e extrair suas partes constituintes[3]. O primeiro passo desse processo consiste em segmentar o programa via técnica de "Slicer" (vide Seção 2.5) em termos de suas variáveis relevantes para abstração (vide Seção 3.2.1). O segundo passo do processo consiste em decompor a fatia do código fonte obtida pelo "Slicer" em uma hierarquia de programas primos.

---

(1) Observe que a partir de agora o termo segmentação é vinculado à técnica de "Slicer", e o termo decomposição, aos programas primos.

(2) Neste trabalho de pesquisa os termos síntese e abstração são intercambiáveis.

(3) As partes identificáveis do programa podem ser de dois tipos para esta dissertação: parte de um programa baseado em seu fluxo de dados e controle, e parte de um programa identificável a partir dos construtores da programação estruturada (seqüência, decisão e iteração).

O processo de síntese visa determinar a função executada (abstração funcional) para cada tipo de primo da hierarquia (seqüência, decisão e iteração). Como este processo de síntese é "Bottom-Up", todos os primos de nível hierárquico mais baixo deverão estar abstraídos quando da determinação da função executada de um programa primo de nível hierárquico mais alto.

A Figura 3.1 apresenta diagramaticamente o modelo de abstração funcional para os processos de análise e de síntese.

Observe que pela Figura 3.1 esses dois processos (análise e síntese) são interdependentes. O segmento encontrado pelo "Slicer" é decomposto em programas primos, os quais serão sintetizados até que todo o segmento do programa alvo esteja abstraído. Neste ponto, o fluxo de controle volta para o processo de análise, permitindo que o programa seja segmentado para outras variáveis de interesse. Este novo segmento terá sua função determinada pelo processo de síntese. Esta sucessão de processos será executada até que todos os segmentos definidos pelas variáveis de interesse estejam abstraídos.

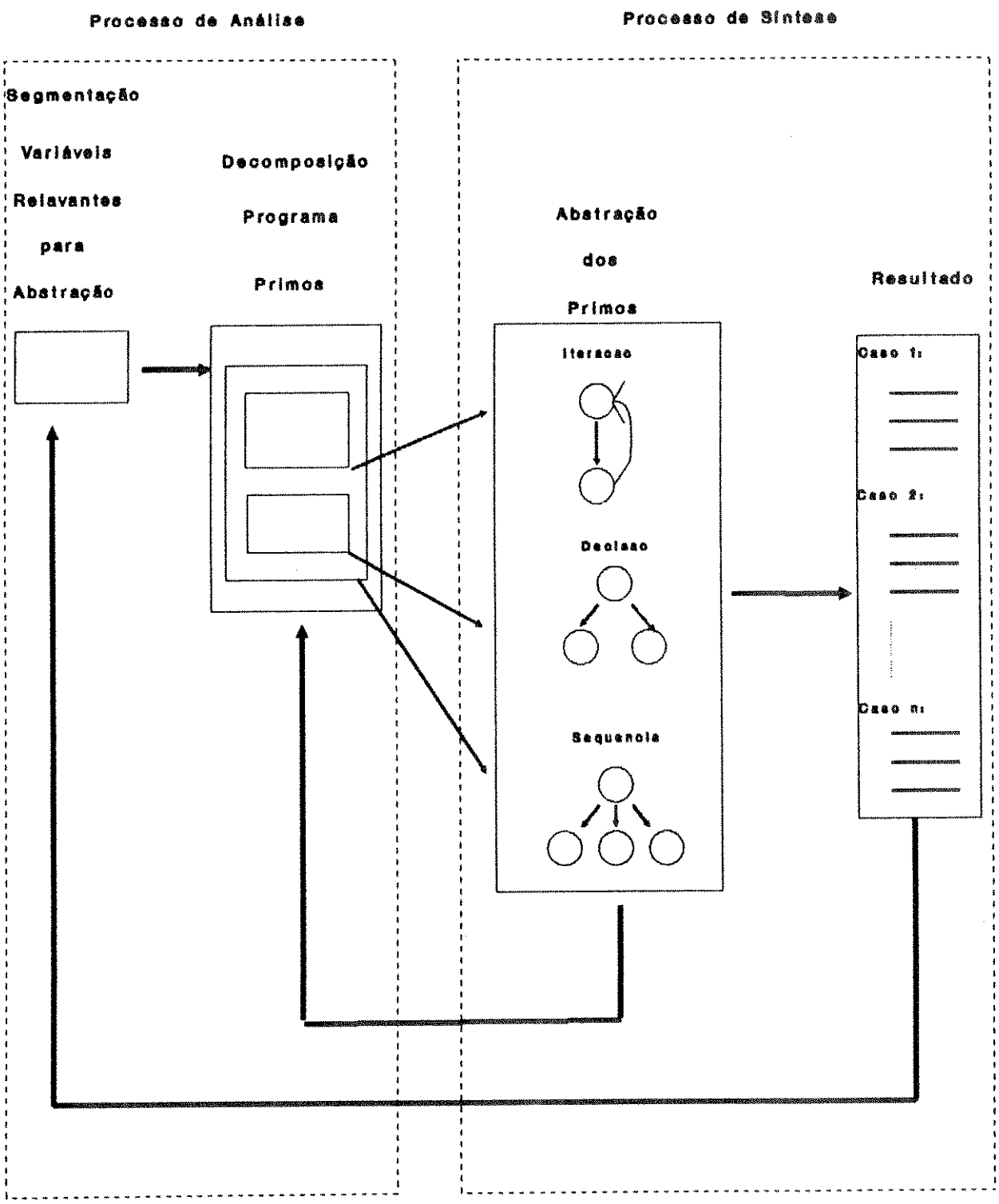


Figura 3.1: Diagrama do Modelo para Abstração Funcional

3.2 Processo de Análise do Programa

Este processo analisa um programa com o intuito de identificação e extração das suas partes constituintes. Ele é formado por dois subprocessos: segmentação do programa em relação às suas Variáveis de Estado (segmentação baseada no fluxo de dados e controle) e decomposição em programas primos (decomposição baseada nos construtores utilizados).

3.2.1 Segmentação do Programa em Função de Suas Variáveis de Estado

Todo programa pode possuir três tipos de variáveis em relação ao seu uso: variáveis de entrada, variáveis intermediárias e variáveis de saída.

As variáveis de entrada representam os argumentos passados para as sub-rotinas e as variáveis globais referenciadas nos comandos. As variáveis de saída podem retornar valores para uma ou mais sub-rotinas (por exemplo, as variáveis que são parâmetros passados por referência e as variáveis globais que recebam valor). Todas as outras variáveis podem ser classificadas como intermediárias.

Porém, esta classificação não é suficiente para capturar a intenção do programa, isto é, o que o programa pretende fazer. Neste sentido, durante este trabalho de pesquisa o autor sentiu a necessidade de estabelecer um novo conceito intitulado de Variáveis de Estado (veja conceito 1 a seguir), as quais são determinantes para a segmentação do programa.

Conceito 1: Variáveis de Estado

Este novo conceito de Variáveis de Estado consiste basicamente na idéia de que o estado final de um programa pode ser definido a partir do conjunto de todas as variáveis que armazenam a intenção computacional do programa. Este conjunto é formado por todas as variáveis globais que recebam valor no programa, pelos parâmetros formais passados por referência que recebam valor, pelas variáveis de retorno do programa (comando "return" na linguagem C) e, finalmente, pelas variáveis referenciadas em comandos de gravação ou impressão. Essas variáveis podem ser formalizadas de acordo com a notação abaixo:

Notação para as Variáveis de Estado
Vs = { v | v é uma variável utilizada em comandos de gravação ou impressão de P } Vp = { v | v é um parâmetro formal passado por referência, cujo valor foi definido em P } Vg = { v | v é uma variável global, cujo valor foi modificado em P } Vr = { v | v é uma variável referenciada em comandos de retorno de algumas linguagens de programação }
Onde: P -> é um programa. v -> é uma variável de P.

Observa-se que qualquer variável de uma sub-rotina, que não esteja de acordo com a notação formalizada acima, torna-se irrelevante num processo de determinação da função do programa. Ou seja, ela pode ser utilizada para auxiliar o desenvolvimento do algoritmo implementado, mas não serviria necessariamente para capturar o significado do programa. Neste sentido, o resultado da abstração funcional do programa somente necessita apresentar as Variáveis de Estado aos usuários, o que não significa, entretanto, que as variáveis intermediárias e de entrada não sejam trabalhadas no processo.

A Figura 3.2 retrata as variáveis de um programa de acordo com o conceito de Variáveis de Estado.

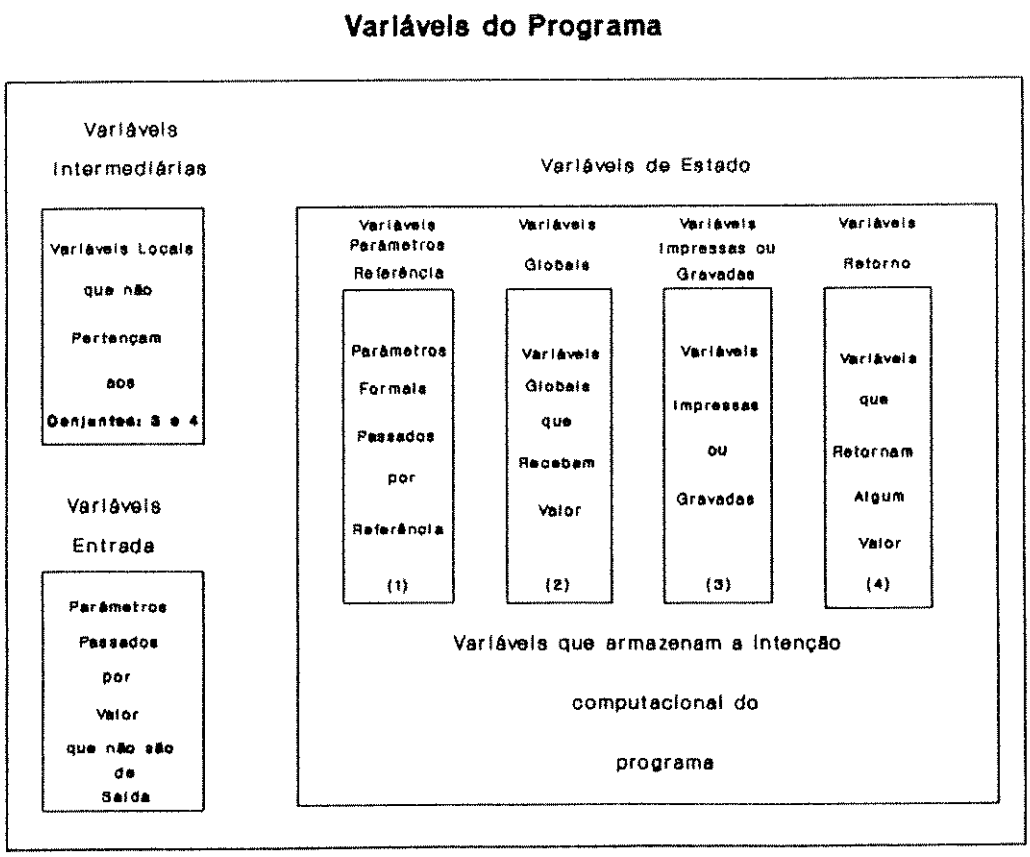


Figura 3.2: Variáveis de um Programa Típico

Cabe salientar que a segmentação do programa em variáveis é fundamental para a abstração de programas que reutilizam a mesma variável em várias tarefas, pois, dependendo do programa, a reutilização de variáveis pode tornar o resultado final da abstração funcional insatisfatório, exigindo outras transformações no programa. Considere o exemplo da Figura 3.3.

```
1  fscanf("%d", &a);
2  fscanf("%d", &b);
3  t = a + b;
4  printf("%d\n", t);
5  t = a - b;
6  printf("%d\n", t);
```

Figura 3.3: Exemplo de Programa que Reutiliza Variável

Uma transformação deste tipo de programa via reengenharia de dados é proposta por Hausler com o objetivo de permitir que os comandos "t = a + b" e "t = a - b " possam ser analisados independentemente [HAUS-90]. Neste contexto, a variável "t" do segundo comando teria seu nome trocado (por exemplo, "t1 = a - b"), permitindo a representação dos dois comandos na abstração funcional final. Porém, o comando "t = a + b" só é relevante para este programa porque o seu resultado é impresso na linha 4, caso contrário, o comando em questão torna-se um código morto.

Uma alternativa à utilização da reengenharia de dados no processo de abstração é a utilização de uma nova estratégia, baseada na técnica de "Slicer" de programas e aplicada ao novo conceito de Variáveis de Estado. Desta forma, o programa é segmentado, utilizando-se a técnica de "Slicer" a partir dos critérios gerados pelas Variáveis de Estado.

A Tabela 3.1 apresenta a formalização para a segmentação de um programa em termos de suas Variáveis de Estado.

Seja: P -> é um programa
v ∈ { Vs U Vp U Vg U Vr}

Sejam os conjuntos:

Saída(P, v): Conjunto de comandos em P, que imprimem ou gravam a variável v.
L : Último comando de P.
N : Saída(P, v) U L

A segmentação do programa para a variável "v" constituirá um conjunto de subpartes do programa que armazena todas as computações relevantes para a Variável de Estado "v". Ou seja,

$$S(v) = \bigcup_{n \in N} Slicer(P, v, n)$$ onde n = L se n ∈ { Vp U Vg U Vr }
n ∈ Saída(P, v) se v ∈ Vs

Tabela 3.1: Segmentação do Programa em Termos de Suas Variáveis de Estado

Esta formalização deve ser seguida para cada Variável de Estado, visando permitir que todos os comandos que afetam estas variáveis possam ser abstraídos.

Considere a Figura 3.4, cujo programa, extraído de [KERN-78], computa o número de linhas, palavras e caracteres de um arquivo de entrada.

```
 1 #define YES 1
 2 #define NO  0
 3 main()
 4 {
 5      int c, nl, nw, nc, inword;
 6      inword = NO;
 7      nl = 0;
 8      nw = 0;
 9      nc = 0;
10      c = getchar();
11      while(c != EOF) {
12           nc = nc + 1;
13           if (c == '\n')
14                nl = nl + 1;
15           if (c == ' ' || c == '\n'|| c == '\t')
16                inword = NO;
17           else if (inword == NO) {
18                inword = YES;
19                nw = nw + 1;
20           }
21           c = getchar();
22      }
23      printf("%d \n", nl);
23      printf("%d \n", nw);
25      printf("%d \n", nc);
26 }
```

Figura 3.4: Programa que Calcula o Número de Linhas, Palavras e Caracteres de um Arquivo de Entrada.

A técnica de "Slicer" foi utilizada para gerar a segmentação do programa em termos de suas Variáveis de Estado (nl, mw e nc), representada nas Figuras 3.5, 3.6 e 3.7. Observe que cada uma delas é um programa completo para um dado critério.

```
 3 main()
 4 {
 5       int c, nl;
 7       nl = 0;
10       c = getchar();
11       while(c != EOF) {
13            if (c == '\n')
14                 nl = nl + 1;
21                 c = getchar();
22        }
23        printf("%d \n", nl);
26 }
```

Figura 3.5: Slicer Aplicado sobre a Variável "nl".

```
 1 #define YES 1
 2 #define NO  0
 3 main()
 4 {
 5      int c, nw, inword;
 6      inword = NO;
 8      nw = 0;
10      c = getchar();
11      while(c != EOF) {
15           if (c == ' ' || c == '\n'|| c == '\t')
16                inword = NO;
17           else if (inword == NO) {
18                inword = YES;
19                nw = nw + 1;
20           }
21           c = getchar();
22      }
23      printf("%d \n", nw);
26 }
```

Figura 3.6: Slicer Aplicado sobre a Variável "nw".

```
 1 #define YES 1
 2 #define NO  0
 3 main()
 4 {
 5      int c, nc;
 9      nc = 0;
10      c = getchar();
11      while(c != EOF) {
12           nc = nc + 1;
21           c = getchar();
22      }
25      printf("%d \n", nc);
26 }
```

Figura 3.7: Slicer Aplicado sobre a Variável "nc".

O programa obtido com a aplicação do "Slicer" para o critério SC(P,23,nl)[4] irá produzir o número total de linhas de um arquivo (Figura 3.5). A aplicação do critério SC(P,24,nw) irá computar o número total de palavras (Figura 3.6). A especificação SC(P,25,nc) irá computar o número de caracteres de um arquivo (Figura 3.7).

Vale ressaltar que o processo de abstração será orientado para obter descrições funcionais (abstrações) das variáveis que explicam a intenção do programa (Variáveis de Estado) e não para todas as variáveis do mesmo. A razão principal para se proceder à segmentação é a diminuição do número de caminhos a serem considerados numa abstração, reduzindo, assim, a complexidade do programa analisado. Uma outra razão é o impedimento à perdas de informações relevantes sobre os comandos que afetam as variáveis do programa.

Considere a segmentação do programa que está na Figura 3.3, a qual será produzida por SLICER(P,3,t) e SLICER(P,6,t), cujos resultados se encontram nas Figuras 3.8 e 3.9, respectivamente.

---

[4] SC representa o Critério para o "Slicer".

```
1 fscanf("%d",&a);
2 fscanf("%d",&b);
3 t = a + b;
```

Figura 3.8: Slicer(P, 3, t)

```
1 fscanf("%d", &a);
2 fscanf("%d", &b);
3 t = a - b;
```

Figura 3.9: Slicer(P, 6, 7)

Pode-se observar que o programa gera dois valores para "t". Se o programa fosse analisado sem a segmentação, o sexto comando ("t = a-b") anularia o efeito produzido pelo terceiro comando ("t = a+b"), não refletindo exatamente a funcionalidade do programa.

3.2.2 Decomposição em Programas Primos

Este subprocesso visa decompor um programa em uma hierarquia de programas primos, os quais representam as unidades básicas de abstração do mesmo. A forma de análise e de abstração de cada primo depende do seu tipo (seqüência, decisão ou iteração).

3.3 Abstração Funcional dos Programas Primos

Nesta seção são apresentadas as técnicas para abstração funcional dos programas primos do tipo decisão, do tipo seqüência ou do tipo iteração. As abstrações de decisões são obtidas pela geração de casos para a parte "then" e para a parte "else" do primo em questão (vide Seção 3.3.1). As seqüências são abstraídas utilizando-se as técnicas de "Trace-Table" e "Trace-Table" condicional (vide Seção 3.3.2), como foi proposto por Hausler [HAUS-90]. Por fim, as iterações têm as suas computações sintetizadas através da resolução das relações de recorrência que estejam em seu corpo (vide Seção 3.3.3), de acordo com proposta de Cheatham [CHEA-79].

A Figura 3.10 ilustra as técnicas utilizadas para abstração de cada programa primo.

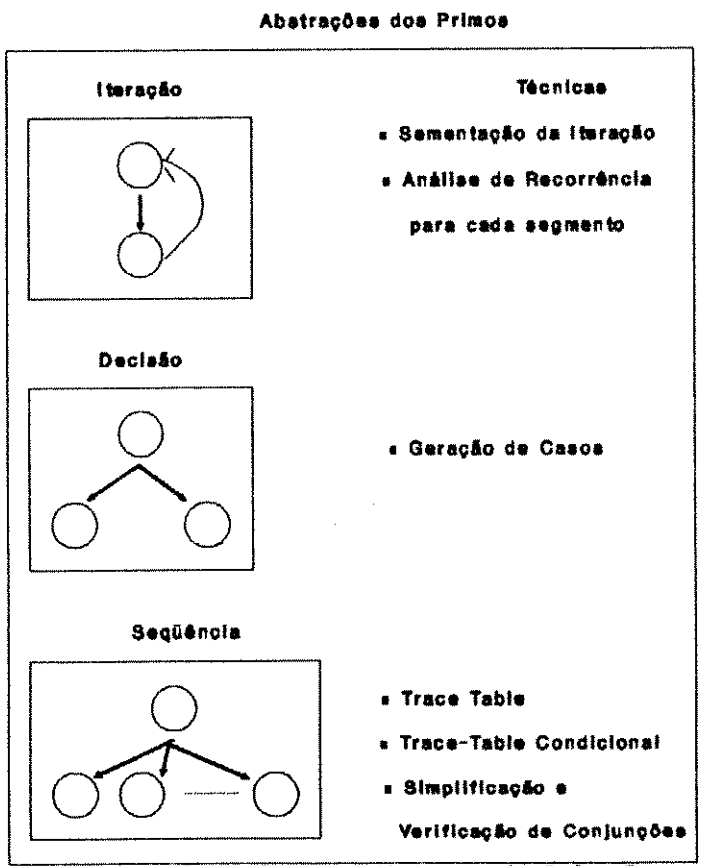


Figura 3.10: Técnicas Utilizadas para Abstração de Cada Programa Primo

Os resultados das abstrações dos programas primos devem ser especificados com a atribuição concorrente, como foi descrito na Seção 2.3. Neste caso, duas sintaxes devem ser utilizadas para capturar o efeito dos programas primos [HAUS-90, MILL-75, LING-79]:

Atribuição Concorrente:

<id>, <id>, ..., <id> := <expr>, <expr>, ..., <expr>
onde: id = identificador (variável)
expr = expressão (efeito do programa sobre a variável)

Atribuição Concorrente Condicional:

(<Condição> -> (Atribuição Concorrente) |
<Condição> -> (Atribuição Concorrente) |
.
.
.
<Condição> -> (Atribuição Concorrente))

Observe que a atribuição concorrente condicional é formada por várias possíveis regras de execução, as quais podem ser chamadas de casos de execução a partir deste ponto da dissertação.

3.3.1 Abstração Funcional dos Comandos de Decisão

A abstração de um comando de decisão pode ser descrita como a geração de "n" possíveis regras de execução, as quais correspondem à soma dos "p" casos existentes na parte "then" com os "q" casos existentes na parte "else" [HAUS-90, BASI-75]. Deve-se observar, contudo, que as condições das primeiras "p" regras de "n" devem ser conjugadas com a condição do comando de decisão "if" e as condições das últimas "q" regras de "n" devem ser conjugadas com a negação da condição do "if".

O comando de decisão deverá ser transformado em regras disjuntas (vide Seção 2.3) para tornar a abstração viável. Por exemplo:

[if p then G else H] = (p -> [G] ¦ ~p -> [H])

Observa-se que as regras disjuntas são mais convenientes do que as regras condicionais, além de serem facilmente obtidas, bastando aumentar cada predicado com a negação de todos os predicados anteriores. O exemplo a seguir ilustra uma transformação para regras disjuntas.

```
if (x > 0) {

     if (x > y)
          z = x;
     else
          z = y;

}
else {

     if (y > 0) {

          if (x < y)
               z = x;
          else
               z = y;

     }
}
```

Comandos transformados:

(x > 0 -> (x > y -> z := x | x ≤ -> z := y) |

x ≤ 0 ^y > 0 -> (x < y -> z := x | x >= y -> z := y))

Distribuindo-se os predicados externos pelas regras internas, obtém-se:

```
(x > 0 ^ x > y          -> z := x |
 x > 0 ^ x ≤ y          -> z := y |
 x ≤ 0 ^ y > 0 ^ x < y  -> z := x |
 x ≤ 0 ^ y > 0 ^ x ≥ y  -> z := y)
```

Pode-se notar que o último predicado é falso e, portanto, pode ser eliminado. A Seção 3.3.2.1 apresenta uma técnica para avaliar se um predicado é factível.

3.3.2 Abstração Funcional de Seqüências

A seqüência é definida como um conjunto de comandos ou abstrações (primos já sintetizados), que são executados incondicionalmente e em ordem. O modelo de abstração funcional requer a consideração de dois tipos de sequências. O primeiro tipo refere-se a um programa primo de mais baixo nível, composto apenas por atribuições. O segundo tipo engloba as seqüências que possuem atribuições e abstrações parciais em seu corpo, obtidas pela abstração de outros primos de nível hierárquico mais baixo. Neste último caso, o corpo da seqüência pode possuir atribuições concorrentes, atribuições condicionais e funções fechadas, que representam abstrações de iterações (vide Seção 3.3.3).

Abstração Funcional de Seqüências de Atribuições

A técnica para execução simbólica de programas conhecida por "Trace-Table" [LING-79] é utilizada no modelo de abstração funcional para determinar a funcionalidade de seqüências que possuam apenas comandos de atribuição, como foi proposto por Hausler [HAUS-90].

Uma "Trace-Table" é uma tabela de equações, na qual cada linha corresponde a um comando seqüencial do programa e cada coluna corresponde a uma variável assinalada nos comandos que se encontram nas linhas da tabela. Cada célula da tabela indica o estado atual da variável da coluna em relação ao comando da linha. Ademais, o subscrito zero deve ser utilizado para simbolizar o estado inicial das variáveis e os subscritos maiores do que zero, para simbolizar os estados subseqüentes. Por exemplo, considere os comandos abaixo:

```
1.   x = x + y
2.   y = x - y
3.   x = x - y.
```

A seguinte tabela seria produzida para a seqüência acima:

Comando	x	y
1. x = x + y	x1 := x0 + y0	y1 := y0
2. y = x - y	x2 := x1	y2 := x1 - y1
3. x = x - y	x3 := x2 - y2	y3 := y2

Vale destacar que cada variável é definida em termos dos valores imediatamente predecessores, bem como as variáveis não definidas nas expressões.

A abstração da funcionalidade desses comandos requer a eliminação de todos os subscritos intermediários da tabela. O processo é bastante simples, começando a partir da última linha da tabela e repetindo-se através de substituições até a primeira linha da tabela. Utilizando-se o mesmo exemplo, tem-se:

```
x3 := x2 - y2                    y3 := y2
   := x1 - (x1 - y1)                := x1 - y1
   := y1                            := x0 + y0 - y0
   := y0                            := x0
```

Como pode ser observado a partir da "Trace-Table", o valor final da variável "x" é o valor inicial de "y" e o valor final de "y" é o valor inicial de "x", o que fundamenta a conclusão de que a funcionalidade deste conjunto de comandos é a troca dos valores das variáveis "x" e "y".

A forma matemática mais conveniente para representar a abstração acima é a atribuição concorrente **(x,y := y,x)**, que define uma expressão matemática para o programa analisado. Esta especificação é mais conveniente do que a utilização de dois comandos para indicar a funcionalidade acima (por exemplo, **x = y; y = x;**), o que não permitiria uma interpretação única.

Considere uma outra forma de implementar a troca entre dois números inteiros:

1. t = x
2. x = y
3. y = t

Estes comandos gerariam a seguinte tabela:

Comando	t	x	y
1. t = x	t1 := x0	x1 := x0	y1 := y0
2. x = y	t2 := t1	x2 := y1	y2 := y1
3. y = t	t3 := t2	x3 := x2	y3 := t2

Derivações:

t3 := t2	x3 := x2	y3 := t2
:= t1	:= y1	:= t1
:= x0	:= y0	:= x0

A funcionalidade é expressa por **t, x, y := x, y, x.**

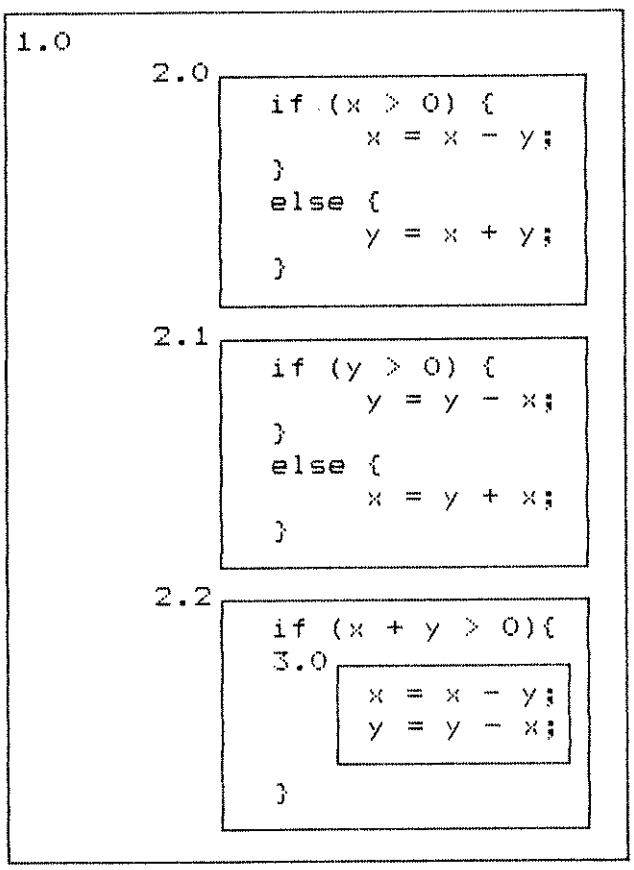
Observe que a expressão final possui uma variável "t", que foi utilizada temporariamente para possibilitar a troca entre as variáveis "x" e "y". Contudo, como não há garantia de que a variável "t" seja utilizada posteriormente com o valor de "x", isto é, definindo outra variável antes de receber qualquer outra atribuição, não é possível eliminá-la desta abstração parcial. Sendo assim, ela será retirada após a abstração funcional completa do programa, assim como todas as variáveis que não forem de estado.

Abstração Funcional de Seqüências de Atribuições Condicionais

A técnica de "Trace-Table" deve ser utilizada para cada possível combinação das condições envolvidas nas atribuições condicionais [HAUS-90], sendo conhecida como "Trace-Table" condicional [LING-79]. Por exemplo, considere os seguintes comandos a serem analisados:

```
if (x > 0) {
     x = x - y;
}
else {
     y = x + y;
}
if (y > 0) {
     y = y - x;
}
else {
     x = y + x;
}
if (x + y > 0) {
     x = x - y;
     y = y - x;
}
```

A decomposição em primos deste programa é ilustrada abaixo:



As funções dos primos internos 3.0, 2.0, 2.1 e 2.2 devem ser derivadas em primeiro lugar, ou seja:

Primo 3.0: x, y := x-y, y-x

Primo 2.0: (x > 0 -> x := x - y ¦
x <= 0 -> y := x + y)

Primo 2.1: (y > 0 -> y := y - x ¦
y <= 0 -> x := y + x)

Primo 2.2: ((x+y) > 0 -> x,y := x-y, y-x ¦
(x+y) <= 0 -> x,y := x, y)

Vale observar que a função do primo 2.2 incorpora a função encontrada para o primo 3.0. Além disso, o primo 2.2 possui em sua abstração a negação da condição encontrada no "if", mesmo que o "if" não tenha a cláusula else, mantendo-se a homogeneidade do processo.

A abstração para o primo 1.0 será obtida pela combinação das possíveis regras existentes nos três primos de nível hierárquico mais baixo, produzindo oito diferentes casos, cada qual envolvendo uma simples "Trace-Table". Pode-se generalizar o número total de combinações a partir da fórmula 2^n, onde "n" indica o número de regras envolvidas.

Como pode ser visto no exemplo a seguir, as condições associadas a cada regra são inseridas em uma nova coluna da tabela. Os subscritos que acompanham estas condições são utilizados para referenciar o estado corrente da variável e dependem do momento de avaliação de cada predicado, devendo assumir valores imediatamente anteriores à sua avaliação.

A derivação das atribuições e condições seguem a geração da tabela, visando à obtenção da funcionalidade dos comandos e das condições de execução. Neste caso, deve-se proceder à substituição inversa das variáveis que possuem os subscritos finais pelas suas definições anteriores. Vale notar que neste processo de derivação poderá ocorrer um caso infactível, ou seja, um caso em que as condições assumirão sempre o valor boleano falso.

Assim, continuando a abstração funcional do exemplo anterior, tem-se:

Notação:

Caso i,j,k: Os índices i, j e k serão utilizados para indicar as regras dos primos 2.0, 2.1 e 2.2 respectivamente. Eles assumirão apenas os valores 1 ou 2, levando-se em conta que cada regra possui apenas dois casos.

Caso 1,1,1:

Comando	condição	x	y
1. x = x - y	x0 > 0	x1 := x0 - y0	y1 := y0
2. y = y - x	y1 > 0	x2 := x1	y2 := y1 - x1
3. x,y = x - y, y - x	x2 + y2 > 0	x3 := x2 - y2	y3 := y2 - x2

Derivações:

Condições:

$$x0 > 0 \wedge y1 > 0 \wedge (x2 + y2) > 0 =$$
$$x0 > 0 \wedge y0 > 0 \wedge (x1 + y1 - x1) > 0 =$$
$$x0 > 0 \wedge y0 > 0 \wedge y0 > 0$$

Atribuições:

$$x3 := x2 - y2 := x1 - y1 + x1 := 2x0 - 2y0 - y0 := 2x0 - 3y0$$

$$y3 := y2 - x2 := y1 - x1 - x1 := y0 - x0 + y0 - x0 + y0 := 3y0 - 2x0$$

Caso 1,1,2:

Comando	condição	x	y
1. x = x - y	$x0 > 0$	$x1 := x0 - y0$	$y1 := y0$
2. y = y - x	$y1 > 0$	$x2 := x1$	$y2 := y1 - x1$
3. x,y = x,y	$x2 + y2 \leq 0$	$x3 := x2$	$y3 := y2$

Derivações:

Condições:

$$x0 > 0 \wedge y1 > 0 \wedge x2 + y2 \leq 0 =$$
$$x0 > 0 \wedge y0 > 0 \wedge x1 + y1 - x1 \leq 0 =$$
$$x0 > 0 \wedge y0 > 0 \wedge y0 \leq 0 = \text{FALSO}$$

Atribuições: Não são necessárias, pois a condição não é factível.

Caso 1,2,1:

Comando	condição	x	y
1. x = x - y	$x0 > 0$	$x1 := x0 - y0$	$y1 := y0$
2. x = y + x	$y1 \leq 0$	$x2 := y1 + x1$	$y2 := y1$
3. x,y = x - y, y - x	$x2 + y2 > 0$	$x3 := x2 - y2$	$y3 := y2 - x2$

Derivações:

Condições:

$$x0 > 0 \wedge y1 \leq 0 \wedge x2 + y2 > 0 =$$
$$x0 > 0 \wedge y0 \leq 0 \wedge y1 + x1 + y1 > 0 =$$
$$x0 > 0 \wedge y0 \leq 0 \wedge y0 + x0 - y0 + y0 > 0 =$$
$$x0 > 0 \wedge y0 \leq 0 \wedge x0 + y0 > 0$$

Atribuições:

$$x3 := x2 - y2 := y1 + x1 - y1 := x0 - y0$$

$$y3 := y2 - x2 := y1 - y1 - x1 := x0 + y0$$

Caso 2,1,1:

Comando	condição	x	y
1. y = x + y	$x0 \leq 0$	$x1 := x0$	$y1 := x0 + y0$
2. y = y - x	$y1 > 0$	$x2 := x1$	$y2 := y1 - x1$
3. x,y = x - y, y - x	$x2 + y2 > 0$	$x3 := x2 - y2$	$y3 := y2 - x2$

Derivações:

Condições:

$$x0 \leq 0 \wedge y1 > 0 \wedge x2 + y2 > 0 =$$
$$x0 \leq 0 \wedge x0 + y0 > 0 \wedge x1 + y1 - x1 > 0 =$$
$$x0 \leq 0 \wedge x0 + y0 > 0 \wedge x0 + y0 > 0$$

Atribuições:

$$x3 := x2 - y2 := x1 - y1 + x1 := 2x0 - x0 - y0 := x0 - y0$$

$$y3 := y2 - x2 := y1 - x1 - x1 := x0 + y0 - 2x0 := y0 - x0$$

Caso 1,2,2:

Comando	condição	x	y
1. x = x - y	$x0 > 0$	$x1 := x0 - y0$	$y1 := y0$
2. x = y + x	$y1 \leq 0$	$x2 := y1 + x1$	$y2 := y1$
3. x,y = x,y	$x2 + y2 \leq 0$	$x3 := x2$	$y3 := y2$

Derivações:

Condições:

$$x0 > 0 \wedge y1 \leq 0 \; x2 + y2 \leq 0 =$$
$$x0 > 0 \wedge y0 \leq 0 \wedge (y1 + x1 + y1) \leq 0 =$$
$$x0 > 0 \wedge y0 \leq 0 \wedge (y0 + x0 - y0 + y0) \leq 0 =$$
$$x0 > 0 \wedge y0 \leq 0 \wedge x0 + y0 \leq 0$$

Atribuições

$$x3 := x2 := y1 + x1 := y0 + x0 - y0 := x0$$

$$y3 := y2 := y1 := y0$$

Caso 2,1,2:

Comando	condição	x	y
1. y = x + y	$x0 \leq 0$	$x1 := x0$	$y1 := x0 + y0$
2. y = y - x	$y1 > 0$	$x2 := x1$	$y2 := y1 - x1$
3. x,y = x,y	$x2 + y2 \leq 0$	$x3 := x2$	$y3 := y2$

Derivações:

Condições:

$$x0 \leq 0 \wedge y1 > 0 \wedge x2 + y2 \leq 0 =$$
$$x0 \leq 0 \wedge x0 + y0 > 0 \wedge x1 + y1 - x1 \leq =$$
$$x0 \leq 0 \wedge x0 + y0 > 0 \wedge x0 + y0 \leq 0 = \text{FALSO}$$

Atribuições: Não são necessárias, pois a condição não é factível.

Caso 2,2,1:

Comando	condição	x	y
1. y = x + y	$x0 \leq 0$	$x1 := x0$	$y1 := x0 + y0$
2. x = y + x	$y1 \leq 0$	$x2 := y1 + x1$	$y2 := y1$
3. x,y = x,y	$x2 + y2 > 0$	$x3 := x2 - y2$	$y3 := y2 - x2$

Derivações:

Condições:

$$x0 \leq 0 \wedge y1 \leq 0 \wedge x2 + y2 > 0 =$$
$$x0 \leq 0 \wedge x0 + y0 \leq 0 \wedge y1 + x1 + y1 > 0 =$$
$$x0 \leq 0 \wedge x0 + y0 \leq 0 \wedge x0 + y0 + x0 + x0 + y0 > 0 =$$
$$x0 \leq 0 \wedge x0 + y0 \leq 0 \wedge 3x0 + 2y0 > 0 = \text{FALSO}$$

Atribuições: Não são necessárias, pois a condição não é factível.

Caso 2,2,2:

Comando	condição	x	y
1. y = x + y	$x0 \leq 0$	$x1 := x0$	$y1 := x0 + y0$
2. x = y + x	$y1 \leq 0$	$x2 := y1 + x1$	$y2 := y1$
3. x,y = x,y	$x2 + y2 \leq 0$	$x3 := x2$	$y3 := y2$

Derivações:

Condições:

$$x0 \leq 0 \wedge y1 \leq 0 \wedge x2 + y2 \leq 0 =$$
$$x0 \leq 0 \wedge x0 + y0 \leq 0 \wedge y1 + x1 + y1 \leq 0 =$$
$$x0 \leq 0 \wedge x0 + y0 \leq 0 \wedge x0 + y0 + x0 + x0 + y0 \leq 0 =$$
$$x0 \leq 0 \wedge x0 + y0 \leq 0 \wedge 3x0 + 2y0 \leq 0$$

Atribuições:

$$x3 := x2 := y1 + x1 := x0 + y0 + x0 := 2x0 + y0$$

$$y3 := y2 := y1 := x0 + y0$$

A função do primo 1.0 pode ser representada através da seguinte especificação:

```
( (x >  0 ^ y >  0 )          -> x,y := 2x-3y, 3y-2x ¦
  (x >  0 ^ y <= 0 ^ x+y > 0) -> x,y :=  x-y ,-x+y   ¦
  (x <= 0 ^ x+y >  0 )        -> x,y :=  x-y , y-x   ¦
  (x >  0 ^ y <= 0 ^ x+y <= 0)-> x,y :=    x , y     ¦
  (x <= 0 ^ x+y <= 0)         -> x,y :=  2x+y, x+y    )
```

O processo contínuo para abstração com "Trace-Table" condicional pode resultar em uma explosão do número de casos gerados, visto que o número de regras condicionais geradas crescem exponencialmente como função do tamanho do programa [LING-79, HAUS-90].

Existem dois caminhos que poderão ser utilizados num método automatizado com o objetivo de minimizar este problema. O primeiro caminho consiste em reconhecer e eliminar casos não factíveis, como, por exemplo, o "caso 2,2,1" mostrado anteriormente. Se este caso não fosse eliminado, outros casos não factíveis seriam gerados quando os primos de mais alto nível fossem analisados [HAUS-90]. A Seção 3.3.2.1 descreve técnicas de padronização e verificação da factibilidade das expressões relacionais.

Mesmo que os casos impossíveis sejam eliminados, é provável que continue existindo um número muito grande de casos gerados. Sendo assim, um segundo caminho seria a transformação dos comandos analisados em uma abstração que forneça um único valor possível [HAUS-90]. Por exemplo, considere os comandos abaixo:

```
if (x >= y)
     z = x;
else
     z = y;
```

Uma abstração funcional destes comandos produziria dois possíveis casos. Contudo, se fosse empregada a abstração "max(_,_)", representando o máximo entre duas entradas, poder-se-ia encapsular a funcionalidade do programa em um simples caso, ou seja,

z := max(x,y)

Esta abstração reduz o número de casos produzidos nas abstrações posteriores e, em adição, aumenta a legibilidade da abstração final. Vale enfatizar, portanto, que a abstração com um simples valor pode representar uma definição matemática complexa, correspondendo a conceitos básicos no domínio de aplicação do programa (por exemplo: mínimo, máximo, valor absoluto, sinal, etc). Ademais, o fato destas abstrações parciais serem formais possibilita a propagação deste resultado em outras "Trace-Tables", apesar de não existir uma solução automatizada para a transformação dos comandos analisados em uma abstração de mais alto nível [HAUS-90].

Em suma, a técnica de "Trace-Table" pode ser utilizada para analisar seqüências e pode representar o resultado como expressões condicionais da seguinte forma:

- **Case(p1, e1, ..., pn, en)**

Neste caso, "ei" (atribuição concorrente) só será executado se, e somente se, "pi" for verdadeiro, assumindo que todos os "pi's" sejam mutuamente exclusivos e exaustivos, isto é,

OR(p1, ..., pn) = VERDADEIRO e AND(pi, pj) = FALSO, para i ≠ j.

Observe que qualquer expressão condicional(*) encontrada ou gerada durante o processo de abstração funcional deve ser padronizada na forma acima, bem como os "pi's" existentes nas expressões. Pode-se destacar as seguintes situações para padronização, de acordo com Cheatham [CHEA-79]:

1) Expressões aninhadas, as quais serão distribuídas pelos outros predicados, por exemplo:

(*) Esta condição é formada por várias expressões relacionais (por exemplo, x < 10) que, por sua vez, são agregadas através dos operadores boleanos "And", "Or" e "Not".

- Case(p1, Case(p2, e2, p3, e3), ...), seria transformado para

  Case(p1 and p2, e2, p1 and p3, e3, ...)

2) Atribuição concorrente, caso particular das expressões condicionais, nas quais o predicado é sempre verdadeiro, por exemplo:

   - **Case(TRUE, e1)**

3.3.2.1 Padronização, Simplificação e Verificação das Expressões Condicionais

Esta subseção descreve as técnicas para padronização, simplificação e verificação das expressões relacionais.

Padronização das Expressões Relacionais

As condições envolvidas nas expressões relacionais podem ser padronizadas para facilitar a simplificação e verificação das mesmas. Neste caso, um conjunto reduzido de expressões relacionais deve ser utilizado:

- L ≤ R -> Le(L, R)
- L = R -> Eq(L, R)
- L ≠ R -> Ne(L, R)

Outras expressões relacionais podem ser padronizadas para um dos tipos acima, de acordo com as regras listadas na Tabela 3.2. Vale observar que esta tabela foi criada pelo autor desta pesquisa a partir da proposta de padronização de Cheatham [CHEA-79].

Símbolos Utilizados:

L : Operando do lado esquerdo da expressão.

R : Operando do lado direito da expressão.

$\in$: Menor número real significativo passivel de representação no computador, tal que: $1 + \in > 1$ e $\in \approx 0$.

Regra 1: Neste caso, L é constante ou uma expressão sintaticamente mais simples do que R. Deve-se inverter os operandos de modo que o operando do lado direito seja mais simples. Esta padronização permite que expressões relacionais possam ser utilizadas para simplificar outras expressões.

	Expressão				Forma Padronizada		
1.i)	L	$\leq$	R	---->	-R	$\leq$	-L
1.ii)	L	$<$	R	---->	-R	$\leq$	-L-1; Se L e R inteiros
					-R	$\leq$	-L-$\in$; Se L e R reais
1.iii)	L	$\geq$	R	---->	R	$\leq$	L
1.iv)	L	$>$	R	---->	R	$\leq$	L-1; Se L e R inteiros
					R	$\leq$	L-$\in$; Se L e R reais
1.v)	L	=	R	---->	R	=	L
1.vi)	L	$\neq$	R	---->	R	$\neq$	L

Regra 2: Os casos abaixo consideram L diferente de constante e, ainda, uma expressão que não é sintaticamente mais simples do que R. Neste caso, as expressões relacionais $>$, $<$ e $\geq$ devem ser transformadas para o operador $\leq$.

2.i)	L	$<$	R	---->	L	<=	R-1; Se L e R inteiros
					L	<=	R-$\in$; Se L e R reais
2.ii)	L	$>$	R	---->	-L	<=	-R-1; Se L e R inteiros
					-L	<=	-R-$\in$; Se L e R reais
2.iii)	L	>=	R	---->	-L	<=	-R

Tabela 3.2: Padronização das Expressões Relacionais

Além da utilização de um conjunto reduzido de operadores, deve-se rearranjar qualquer igualdade (L = R) de maneira que L seja um potencial candidato para ser trocado por R nas demais expressões. Para que a expressão resultante seja mais simples, R deve ser sintaticamente menos complexo do que L, não contendo qualquer subexpressão que contenha L [CHEA-79].

Por exemplo, Eq(V[i], x) está de acordo com a forma padrão descrita acima, podendo-se substituir V[i] por x nas expressões simbólicas que dependerem desta condição.

Simplificação e Verificação das Expressões Condicionais

A criação de uma condição de execução de um novo programa primo gerado faz-se necessária quando duas abstrações de programas primos precisam ser compostas(6). Esta condição é obtida pela conjunção(7) das condições envolvidas em cada abstração dos programas primos (vide Seção 3.3.2). Neste caso, é fundamental proceder à verificação da factibilidade da nova condição gerada e, também, à sua simplificação.

Segundo Buchberger [BUCH-82], "o problema de simplificação tem dois aspectos: obter um objeto equivalente simplificado e computar uma única representação para objetos equivalentes". Entretanto, nesta pesquisa é enfatizado apenas o aspecto de simplificação das expressões condicionais, sem haver uma preocupação com a obtenção de uma única representação para as mesmas. Vale observar que a verificação da factibilidade é efetuada durante o processo de simplificação.

A técnica de simplificação é totalmente baseada no trabalho de Cheatham [CHEA-79]. Esta técnica tem como pré-requisito a padronização das expressões boleanas na Forma Normal Conjuntiva (FNC), "pois ela é a maneira

(6) Observe que a composição das abstrações funcionais de dois programas primos dar-se-á pela composição das atribuições conconcorrentes simples ou condicionais.

(7) O termo conjunção é empregado para indicar a união lógica de duas ou mais expressões condicionais.

mais natural e eficiente para descrever as condições de execução de um determinado caminho do programa" [CHEA-79].

A FNC não admite que os operandos de uma conjunção sejam conjunções (boleano "AND") e que os operandos de uma disjunção (boleano "OR") sejam conjunção ou disjunção. Contudo, as negações devem ser inseridas nas expressões "And" e "Or" para posterior absorção nas situações possíveis. Por exemplo, se a negação é aplicada a uma constante, às expressões relacionais ou ao boleano "Not", ela deve desaparecer.

A FNC neste trabalho de pesquisa é construída de tal maneira que as Igualdades (Ii), os Outros Operadores Relacionais (Oi) e as Disjunções (Di) são separados em 3 conjuntos, de acordo com a proposta de Cheatham [CHEA-79]. Esta separação é realizada para que se possa aproveitar a propriedade de cada tipo de conjunto, com o intuito de simplificar e verificar casos triviais de infactibilidade. Neste contexto, tem-se:

And(I1, ..., In, O1, ..., Om, D1, ..., Dl), onde:

Ii: representa relação de igualdade (por exemplo, L = R)

Oi: representa relação de desigualdade (por exemplo, $L \neq R$, $L \leq R$)

Di: representa uma disjunção (por exemplo, Or(d1, d2, ..., dn), sendo que cada di é formado por uma igualdade ou uma desigualdade.

Todas as condições do programa devem ser transformadas para FNC, sendo elas relacionadas a comandos "if" ou a comandos "repeat-until", antes que se inicie o processo de obtenção da abstração. O Capítulo 4 descreve em detalhes o momento em que ocorre esta padronização e o algoritmo utilizado.

Cada nova conjunção obtida durante a abstração requer a análise das igualdades, a posterior simplificação das outras unidades relacionais e, finalmente, a simplificação das disjunções.

Observando-se a simbologia adotada abaixo, deve-se aplicar os passos listados imediatamente após.

Simbologia:

C: Conjunção atual. Indica as condições que deverão ser satisfeitas para que as fórmulas simbólicas, associadas a um conjunto de variáveis, sejam válidas. A forma utilizada para C agrega os três conjuntos, ou seja:

And(I1, ..., Ie, O1, ..., Ou, D1, ..., Dd), onde:

Ii, Oi e **Di** são, respectivamente, os conjuntos igualdades, outras unidades relacionais e disjunções.

B: Nova conjunção. Indica que um novo comando ou expressão condicional foi encontrado, devendo este ser conjugado à **C.**

Os seguintes passos extraídos de [CHEA-79] serão aplicados sobre **C** e **B**, visando obter uma nova conjunção simplificada e verificada:

Passo 1: Operações com as igualdades **I1, ..., Ie** de **C.**

Estas igualdades são substituídas apropriadamente em B, isto é, para cada Ii, que possui a forma Eq(Lj, Rj), troca-se cada ocorrência de Lj em B, se existir, por Rj. Se estas operações de substituição reduzirem B para a cláusula verdadeira, B pode ser ignorada, não devendo ser conjugada à C, se B é reduzida para falso, a conjunção inteira (C com B) deve ser desprezada, sendo o caso considerado infactível. Por exemplo:

B = Le(L, 0)

Ii = Eq(L, 10)

Substituindo Rj (10) de Ii em Lj de B, tem-se:

Le(10, 0)

Esta substituição produz uma condição infactível, devendo a conjunção de C com B ser desprezada.

Passo 2: Caso B seja uma igualdade.

Neste caso, B deve ser substituída nas conjunções de C, ou seja, para cada Lj igual a L de B, deve-se substituir Lj por R de B. Cabe eliminar qualquer conjunção que seja reduzida para verdadeiro e, caso contrário, considerar a conjunção de C com B infactível. Finalmente, se as conjunções C e B não forem comparáveis, B deve ser incluída no conjunto das igualdades de C. Por exemplo:

B = Eq(L, 0)

Ii = Eq(L, 10)

Substituindo R (0) de B em Lj de Ii, tem-se:

Eq(10, 0)

Esta substituição produz uma condição infactível, devendo a conjunção de C com B ser desprezada.

Passo 3: Caso B seja uma outra unidade relacional.

Deve-se executar adicionalmente os seguintes passos:

Passo 3.a:

B é conjugado com cada unidade relacional (O1, ..., Ou) e, também, com cada igualdade de C, podendo-se obter vários resultados:

- O resultado da conjunção é falso. Por exemplo,
  Oj = Le(L, A) e B = Le(-L, -(A+1)) ou
  Ij = Eq(L, R) e B = Ne(L, R)
  então, toda a conjunção deve ser eliminada.

- A conjunção de B com algum Oj pode produzir uma outra relação de desigualdade. Considere os seguintes exemplos:
  1. Oj = Le(L, A) e B = Ne(L, A) produz Le(L, A-1); l e a inteiros.
  2. Oj = Le(L,10) e B = Le(L, 0) produz Le(L, 0)
  3. Oj = Le(L,10) e B = Eq(-L,-10) produz Eq(L,10)

Nestes casos, Oj e B são eliminados e a nova relação produzida é adicionada à C.

- As duas conjunções são incomparáveis, devendo B ser adicionada à C.

  Por exemplo:

  Oj = Le(L, A) e B = Ne(L, G)

  Neste caso, Oj e B são incomparáveis.

Passo 3b:

B é analisada contra cada disjunção (D1, ..., Dd), aplicando-se novamente os passos de 1 a 3.a. Por exemplo:

B = Le(L, A)

Dj= Or(Le(-L,-A), d2, ..., dn)

Resolvendo-se B e Dj, tem-se:

Or(Eq(L, A), d2, ..., dn), que deve substituir Dj.

Passo 4: B é uma disjunção.

Todas as outras conjunções de C (conjuntos I e O) devem ser resolvidas contra B e, então, adicionadas à C, sendo que B pode ser reduzida a uma simples conjunção. Por exemplo:

Bj = Or(Le(L, R), Ne(A, 10))

Oi = Le(L, R-1)

Resolvendo-se Le(L, R) de Bj contra Oi, tem-se:

Bj = Ne(A, 10)

Bj foi reduzido para uma conjunção.

A simplificação é importante para que a abstração gerada seja mais concisa, em termos das condições envolvidas nos casos de execução, e para verificar se um determinado caso é factível. Vale adicionar que apenas os

casos mais triviais são passíveis de verificação automática, pois a decisão automática é impossível para todos e quaisquer casos via provas dos teoremas de resolução(8) [KING-76].

3.3.3 Abstração Funcional para Iteração

Esta seção descreve uma técnica para análise e abstração do construtor para iteração, pois é necessário que a iteração seja entendida separadamente para posterior composição, visando à realização de uma análise completa de um programa. Desta forma, o "loop" será considerado, no processo de abstração, como uma função fechada, que produz apenas um caminho no fluxo de controle do programa, se o construtor utilizado for o "repeat-until".

A fundamentação teórica desta análise e abstração possui dois aspectos principais. O primeiro está relacionado ao processo de análise do "loop". Nesta análise, a técnica de "Slicer" de programas é utilizada para quebrar o "loop" em vários elementos, visando facilitar a obtenção da abstração da iteração. O segundo aspecto está relacionado ao processo de abstração e consiste na solução de relações de recorrência.

3.3.3.1 Técnica para Segmentação da Iteração

O modelo proposto por esta dissertação considera o programa, no caso, o "loop", como sendo constituído de vários subsegmentos, que deverão ser entendidos isoladamente antes da compreensão do inter-relacionamento entre eles. Sendo assim, cada parte pode ser considerada como um subproblema que deve ser compreendido e combinado para se tentar buscar o entendimento de cada função computada pelo "loop". Estas

---

(8) O método de resolução é um procedimento parcial que tenta decidir se os argumentos lógicos dos predicados de primeira ordem são factíveis ou não [KING-76].

partes são os elementos computacionais básicos investigados no Capítulo 2 (isto é, "loop" básico, computações independentes e filtros).

A técnica chave para produzir todas as segmentações relacionadas aos componentes básicos chama-se "Slicer", que, por definição, produz um segmento mínimo do código que afeta uma ou mais variáveis de interesse.

Estas segmentações resultam da aplicação da técnica de "Slicer" sobre dois conjuntos de variáveis que devem ser criados. O primeiro conjunto contém as variáveis responsáveis pelo controle computacional ("loop" básico), isto é, as variáveis que são referenciadas nas condições de término do "loop". O segundo conjunto contém todas as outras variáveis assinaladas dentro do corpo do "loop" (variáveis assinaladas nas computações independentes).

A formação destes dois conjuntos deve ser seguida pelo reconhecimento das funções executadas pela iteração. Neste caso, o algoritmo para a técnica de "Slicer" apresentado no Capítulo 2 é executado para cada variável do primeiro conjunto (variáveis de controle), objetivando remover os comandos que afetam os predicados e, assim, conseguir os elementos necessários para se tentar determinar o número de iterações do "loop". O mesmo processo se repete para cada variável do segundo conjunto com o objetivo de separar os diversos cálculos da iteração.

Observa-se que os filtros também são extraídos ao afetarem o comportamento das variáveis estudadas, quando da retirada das computações independentes e do "loop" básico. Portanto, deve-se considerar que as computações são removidas com os respectivos filtros.

O próximo passo é analisar cada computação encontrada, visando desenvolver a sua relação de recorrência. Porém, a análise destas computações não é uma tarefa trivial, sendo impossível analisar automaticamente toda e qualquer computação extraída do "loop". Isto se deve ao fato de que, por exemplo, o programa pode fazer reuso indiscriminado de variáveis, o que tornaria cada computação extraída muito mais complexa, ou, no pior dos casos, todo o "loop" poderia ser extraído como uma única

computação para o propósito da técnica de resolução das relações de recorrência.

3.3.3.2 Técnica para Resolução das Relações de Recorrência

Uma série de expressões que refletem as diversas computações do "loop" é obtida do processo de segmentação com o "Slicer". Neste caso, dois problemas mais gerais deverão ser resolvidos: a resolução do "loop" básico e das computações independentes.

Abstração Funcional do "Loop" Básico

O "loop" básico representa o seu controle computacional, isto é, as condições de parada e os comandos que afetam cada condição. A sua abstração funcional tem como objetivo a obtenção de uma expressão simbólica para o número de ciclos (IL) que o "loop" deverá executar, o qual pode ser expresso genericamente por:

$$IL = Primeiro\ (j,\ 1,\ limite,\ p1(j)\ or\ \ldots\ or\ pn(j))$$

cuja semântica é o primeiro j, tal que $1 \leq j \leq limite$ e $pi(j) = verdadeiro\ (1 \leq i \leq n)$.

Cabe ressaltar a suposição de que o "loop" sempre termine e que "pi" seja uma determinada condição que controle o término do "loop".

Esta expressão genérica deve ser resolvida em função dos comandos que afetam o valor das variáveis envolvidas nas condições. Nesta situação, deve-se obter os seguintes elementos básicos para cada variável referenciada nas condições de término do "loop" em questão:

i) Valor inicial da variável analisada;
ii) Expressão que modifica o valor da variável;
iii) Operador condicional utilizado no predicado.

Observe que as condições envolvidas estarão padronizadas na Forma Normal Conjuntiva (vide Seção 3.3.2.1), ou seja,

(And(I1, ..., In, O1, ..., Om, OR(D11,...,D1i),...,OR(Dj1,..., Djl))

A solução automática irá depender do tipo de comando de incrementação que está sendo utilizado, do tipo de condição e do operador relacional empregado ("≤", "=" e "≠", vide Seção 3.3.2.1). Desta forma, se a condição e o comando que afetam uma determinada variável "X" obedecerem ao seguinte padrão:

. Condição : X operador1 expr2
. Incremento: X = X operador2 expr3

pode-se ter uma série de variações com relação aos operadores 1 e 2 do padrão acima:

i) Se o operador de comparação é "≤" e a expressão incremento é uma progressão aritmética, a solução automática é:
N = (1 + Floor ((expr2-X) / expr3))
O número de iterações é definido em termos das expressões envolvidas na condição de parada (expr2), na incrementação (expr3) e no valor inicial da variável X.

ii) Se o operador de comparação é "≠" e a expressão incremento é uma progressão aritmética, a solução automática é:
N = (Ceiling(expr2-X) / expr3)

iii) Se a expressão incremento é uma progressão geométrica, deve-se reduzir a expressão para soma com uso de logaritmo e, então, aplicar uma das soluções anteriores.

Após a análise de todas as condições de parada, deve-se aplicar a seguinte função para que a solução final seja encontrada:

```
IL = Mínimo(Solução para I1, ..., Solução para In,
            Solução para O1, ..., Solução para Om,
            Máximo(Sol. para D11, ..., Sol. para D1i),
               .
               .
               .
            Máximo(Sol. para Dj1, ..., Sol. para Djl))
```

O segmento de programa abaixo exemplifica a resolução algébrica do número de ciclos que um "loop" deve executar:

```
do {
     i = i + 2;
} while (i ≤ 10)
```

O valor da variável "i" pode ser expresso na forma recorrente:

$$i_{k+1} = i_k + 2$$

A condição de finalização do "loop" é derivada do segmento de programa acima:

$$i_k > 10$$

A consideração desta condição de finalização do "loop" e da relação de recorrência anterior permite a obtenção da seguinte expressão para o número de iterações:

$$IL = (1 + floor((10 - i_0) / 2))$$

O valor inicial da variável "i" deve ser utilizado para simplificar a fórmula acima, se ele puder ser determinado.

A obtenção do número de iterações do "loop" é fundamental neste método de abstração pois, sempre que possível, as expressões recorrentes serão resolvidas levando-se em conta o número total de iterações. Vale enfatizar que apenas as expressões que simbolizem progressões aritméticas e

geométricas serão resolvidas automaticamente na implementação do modelo de abstração funcional (Capítulo 4).

Abstração Funcional das Computações Independentes

A abstração funcional das várias computações independentes existentes no corpo do "loop" é realizada mediante o desenvolvimento das relações de recorrência de cada computação. Por exemplo, seja X_K o valor de uma variável associada a uma determinada expressão de recorrência antes da execução do k-ésimo ciclo. Assumindo-se que exista mais um ciclo, considere X_{K+1} o valor da mesma variável após o k-ésimo ciclo. Neste contexto, pode-se desenvolver uma relação de recorrência, se os elementos X_K, X_{K+1}, X_1 (valor inicial de X) e IL forem conhecidos.

Contudo, a resolução destas recorrências nem sempre é uma tarefa simples, fazendo-se necessário definir quais tipos de recorrência serão suportados pelo método. Vale adicionar que as relações de recorrência listadas a seguir estão baseadas no trabalho de Cheatham [CHEA-79].

i) Relação de recorrência invariante

Na relação de recorrência invariante, o valor da variável definida permanecerá constante durante a execução do "loop". Em outras palavras, o resultado para uma variável "X" no ciclo "K" é igual ao resultado obtido no ciclo "K+1", isto é, X_{K+1} = X(k). A solução para esta relação de recorrência é "$X_K = X_1$" [CHEA-79].

ii) Relação de recorrência com parâmetro variável

A relação de recorrência com parâmetro variável não depende do valor anterior da variável nela definida. Neste caso, o resultado da variável no ciclo "K+1" depende de um parâmetro variável, isto é,

$X_{k+1} = h(j)$, onde j depende do número de ciclos.

Considere os três exemplos a seguir:

```
While(i ≤ n) {          While(u ≤ r) {          While(i ≤ 100) {
     x = y[i];               y = sin(u);             fscanf("%d", &y)
        .                       .                       .
        .                       .                       .
        .                       .                       .
}                       }                       }
```

Nestes exemplos, o valor final das variáveis consideradas depende do último valor assumido pelo parâmetro, o que leva à seguinte relação de recorrência:

$X_k(j) = h_{k-1}(j)$, onde h_{k-1} representa uma variável ou função e j é o parâmetro variável dependente de k [CHEA-79].

iii) Relação de recorrência simples

A relação de recorrência simples simboliza somas, produtos, divisões, etc. O valor da variável definida na expressão, no ciclo k+1, depende diretamente do valor da mesma variável no ciclo anterior, com todas as outras variáveis referenciadas na expressão permanecendo constantes [CHEA-79]. Por exemplo,

. Soma:

$X_{k+1} = X_k + h(k)$, cuja solução é

$X_k = X_1 + Soma\_Finita(j, 1, k-1, h(j))$

A função "soma_finita" representa um único valor, o qual é obtido pela soma h(1) + h(2) + ... + h(k-1)

. Produto

$X_{k+1} = X_k * g(k)$

$X_k = X_1 * Produto\_Finito(j, 1, k-1, g(j))$

Observa-se que todas estas expressões podem ser funcionalmente compostas com outras expressões no transcorrer do processo de abstração.

iv) Relação de recorrência simultânea

Na relação de recorrência simultânea existe uma dependência entre duas ou mais relações de recorrência, as quais devem ser hierarquizadas de maneira que o nível mais baixo da hierarquia possua apenas relações simples [CHEA-79]. Este procedimento deve ser seguido primeiro pela análise das relações independentes e depois pela análise das relações dependentes. Considere o seguinte programa:

```
j = 0;
X = 0;

do {
     X = X + Y[j];
     j = j + 1;
} while (j ≤ 10)
```

Neste exemplo, tem-se as seguintes relações de recorrência:

$$X_{k+1} = X_k + Y[j_k]$$
$$j_{k+1} = j_k + 1$$

A primeira equação depende do resultado obtido para a segunda equação que, por sua vez, forma o "loop" básico, juntamente com a condição de parada.

O primeiro passo é resolver a relação de recorrência do "loop" básico, ou seja:

$$j_{k+1} = j_k + 1 \ \hat{} \ j_k > 10 \ \hat{} \ j1 = 0.$$

que deriva:

$$IL = (1 + floor((10 - 0) / 1) = 11$$

O segundo passo é resolver a relação para X_k, que depende da solução encontrada para i_k; neste caso:

$$X_k = X_1 + Soma\_Finita(j, i_1, IL-1, Y[j])$$

Substituindo os valores de "X1", "IL" e "i_1", tem-se:

$$X_k = soma\_finita(j, 0, 10, Y[j])$$

Vale adicionar, contudo, que as relações de recorrência cíclica (isto é, uma variável "v" pode depender de "w", a qual depende de "v") não serão abordadas automaticamente.

v) Relação de recorrência condicional

A relação de recorrência condicional depende de alguma condição para ser executada. Esta condição é o comando "if" de uma linguagem de programação que, dentro de um "loop" estruturado, receberá a denominação de filtro. Vale esclarecer que o filtro apenas restringe os valores computados pela relação de recorrência em análise.

A solução deste caso envolve a resolução da relação de recorrência, partindo-se depois para a limitação dos dados de entrada via aplicação da função condicional "cond". Esta função possui a seguinte sintaxe:

Cond(Condição, ação da parte "then", ação da parte "else").

Considere o programa abaixo, o qual soma todos os números positivos de um vetor:

```
1. do {
2.    if (x[i] > 0.0) {
3.          soma = soma + x[i];
4.    }
5.    i = i + 1;
6. } while (i ≤ n)
```

Neste exemplo, tem-se uma relação de recorrência condicional e dependente (linhas 2 e 3) e uma expressão recorrente simples (linha 5). Resolvendo a relação que afeta a condição, obtém-se:

$$IL = (1 + floor((n - i1) / 1) = 1 + n - i1$$

A solução para a relação de recorrência da linha 5 é ilustrada abaixo:

$$soma_k = soma_1 + Soma\_Finita(j, i_1, 1 + n - i_1, x[i])$$

Compondo este resultado com o filtro (linha 2), obtém-se:

$$soma_k = soma_1 + Soma\_Finita(j, i_1, 1+n-i_1, cond(x[i] > 0.0, x[i])$$

vi) Forçando uma Solução para Resolução de Relações de Recorrência

Se a resolução de uma determinada relação de recorrência não for possível, pode-se forçar uma solução baseada em funções recursivas [CHEA-79]. Por exemplo, considere uma variável "X", que tem inicialmente o valor "X1", e uma expressão de indução definida como:

$$X_{k+1} = f(X_k, k)$$

A solução desta relação requer a introdução de uma nova função recursiva "XR", cuja finalidade é capturar o efeito do "loop" para uma determinada variável. A função "XR" é associada à função "Lambda", visando armazenar o valor inicial da relação. Desta forma, define-se a seguinte sintaxe:

$$XR(j) = Lambda(j, cond(j=1, X_1, f(XR(j-1), j-1)))$$

Esta função incorpora o valor inicial e o novo valor de "X" para cada ciclo "j". Observa-se que a solução para "X" é a mesma solução encontrada para XR, ou seja, X_K = XR(K).

Considere o programa abaixo que calcula o fatorial da variável "Y":

```
fat = 1;
do {
     fat = fat * y;
     y   = y - 1;
} while (y ≥ 1);
```

Neste exemplo, tem-se as relações de recorrência:

$$y_{K+1} = y_K - 1 \qquad \wedge\ y < 1$$
$$fat_{K+1} = fat_K * y_K$$

Apesar destas duas relações serem passíveis de resolução direta, uma solução recursiva é ilustrada logo abaixo:

```
XR(j) = Lambda(j, cond(j=1, 1, XR(j-1)*j))
```

Esta representação não fornece o valor simbólico para a variável analisada; porém, ela mostra o comportamento do "loop" sobre a mesma, sem criar casos adicionais na abstração final. Vale esclarecer, ademais, que esta representação deverá ser utilizada apenas em último caso pois, antes que ela seja considerada, o usuário tentará fornecer uma abstração que capture o efeito da expressão de recorrência em análise. Tal abstração será uma função que corresponda aos conceitos básicos no domínio de aplicação do programa.

Seja o programa listado logo abaixo:

```
while(i <= 10) {
     while (j <= 10) {
          if (i == j) {
               a[i][j] = 1;
          }
          else {
               a[i][j] = 0;
          }
}
```

Uma abstração por meio de uma função fechada para este programa poderia ser, por exemplo, a seguinte expressão:

```
a = Array(j1, j2, 10, 10, cond(j1=j2, 1, 0))
```

Esta expressão significa um "array" de dimensão 10 por 10, cujos elementos indexados por "j1" e "j2" obedecem à expressão "cond(j1=j2,1,0)", ou seja, "a" é uma matriz identidade.

3.3.3.3 Resumo da Análise e Abstração Funcional da Iteração

A técnica de análise e abstração da iteração procura derivar expressões que capturem o seu efeito sobre as suas variáveis. Neste sentido, a iteração deverá ser dividida em suas partes constituintes para facilitar o desenvolvimento das relações de recorrência. A fundamentação teórica desta técnica de análise e abstração está centrada em 5 premissas:

1. A partir de um programa é possível identificar as suas partes constituintes, quais sejam: computações independentes, "loop" básico e filtros.

2. Cada parte deve ser mais fácil de ser entendida do que o todo.

3. O "Slicer" pode ser utilizado como uma técnica de identificação das partes.

4. Uma porcentagem razoável das relações de recorrência implementa casos mais simples, como soma, produto, contador, máximo, mínimo, etc.

5. Mesmo que não seja possível analisar automaticamente a iteração, pode-se proceder à solução manual, mais fácil de ser obtida, se ela for subdividida em partes menores.

3.4 Considerações Finais

Este capítulo descreveu um modelo de abstração funcional de programas, o qual considera um programa de computador como sendo composto por partes e tenta entender as operações de cada uma, bem como as inter-relações entre elas, com o intuito de desenvolver uma abstração funcional do todo. De um ponto de vista mais abstrato, estas partes podem ser consideradas subproblemas passíveis de extração e síntese.

Este modelo é constituído por dois processos distintos. O primeiro processo visa analisar um programa com o objetivo de empregar o paradigma de dividir para conquistar. Neste processo, o programa é segmentado com a técnica de "Slicer" em termos de suas Variáveis de Estado (conceito definido nesta dissertação na Seção 3.2.1), cada segmento encontrado é decomposto em uma hierarquia de programas primos, que devem ser abstraídos.

O segundo processo determina a função executada por cada programa primo da hierarquia de acordo com o seu tipo (seqüência, decisão ou iteração), segundo uma estratégia "Bottom-Up".

A abstração de um primo do tipo decisão é obtida pela geração de "n" possíveis regras de execução, as quais correspondem à soma dos "p" casos existentes na parte "then" com os "q" casos existentes na parte"else". As técnicas de "Trace-Table" e "Trace-Table" condicional são utilizadas para a abstração de seqüência. Ademais, às duas técnicas anteriores foi acrescida de uma técnica de simplificação e verificação das condições encontradas durante o processo de abstração de seqüências e decisões.

Por fim, a abstração funcional da iteração é obtida pela segmentação da iteração em termos dos seus elementos computacionais básicos ("loop" básico, computações independentes e filtros) e posterior resolução das relações de recorrência encontradas nestes elementos computacionais.

O resultado final da abstração funcional é expresso em termos de possíveis casos de execução do programa, os quais representam o efeito do

programa sobre as suas Variáveis de Estado. Cada caso é composto por uma condição de execução (domínio) e uma série de atribuições concorrentes.

Capítulo 4

Arquitetura e Implementação da Ferramenta de Abstração Funcional

Este capítulo descreve a arquitetura e os algoritmos utilizados para implementar o modelo de abstração funcional apresentado no Capítulo 3. Neste sentido, a Seção 4.1 introduz a arquitetura da ferramenta, a linguagem de computação do programa alvo para abstração e os requisitos básicos do processo. A Seção 4.2 apresenta o subsistema da ferramenta de abstração responsável pela obtenção e manutenção da representação intermediária do programa. A Seção 4.3 descreve o processo de segmentação do programa alvo. Por fim, a Seção 4.4 detalha o processo e os algoritmos utilizados para derivar as abstrações funcionais de cada parte identificada no processo de segmentação.

4.1 Arquitetura da Ferramenta de Abstração Funcional (FAF)

Segundo o modelo descrito no Capítulo 3, existem dois processos principais que devem integrar a FAF: processo de análise do programa, ou seja, sua segmentação e decomposição, e processo de abstração de cada parte do programa encontrada no processo anterior.

O processo de análise visa segmentar o programa alvo em termos de suas Variáveis de Estado (vide Seção 3.2.1) e decompor cada segmento em uma hierarquia de primos. Por sua vez, o processo de abstração objetiva gerar a função executada por cada primo da hierarquia, visando determinar o efeito do programa alvo sobre as suas variáveis.

A Figura 4.1 ilustra os dois principais processos executados pela FAF.

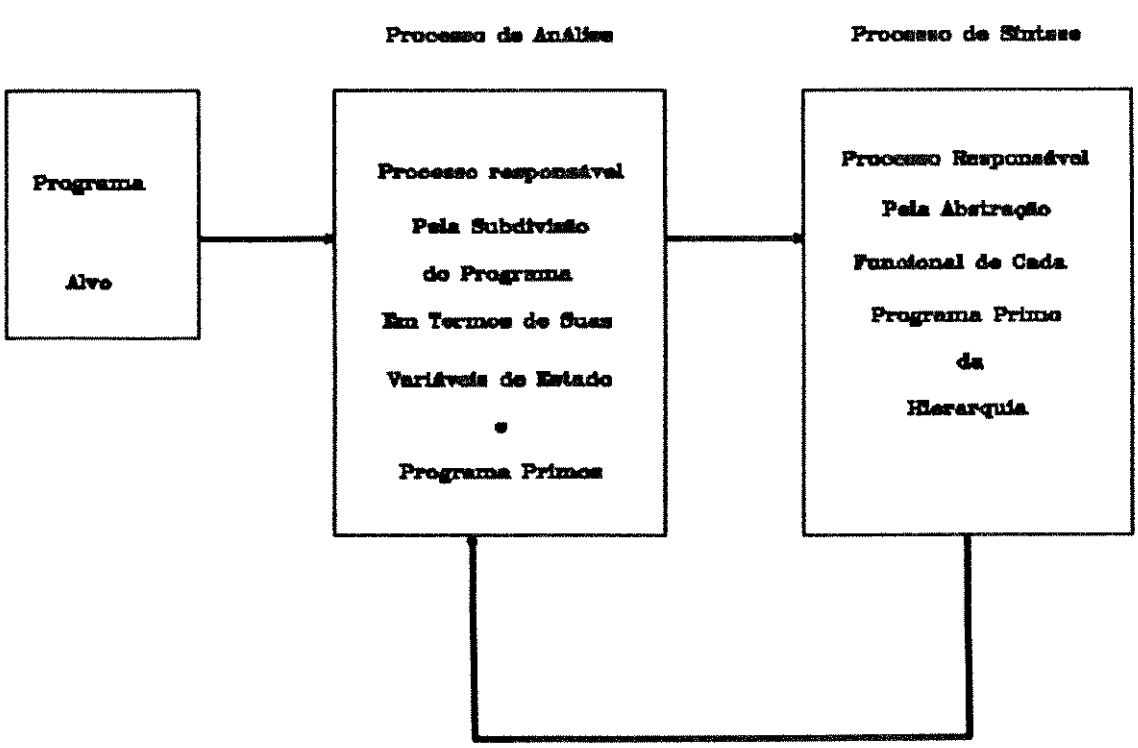


Figura 4.1: Principais Processos Executados pela FAF

Tendo como base os objetivos do processo de análise, o autor desta pesquisa estabeleceu uma forma intermediária que caracterizasse o programa alvo, objetivando possibilitar a utilização do algoritmo de "Slicer" apresentado no Capítulo 2. Neste contexto, o processo em questão foi estendido para suportar duas funções básicas:

1.1) Geração de uma forma intermediária que corresponda ao programa alvo;

1.2) Segmentação do programa alvo, representado pela forma intermediária, e decomposição do mesmo em programas primos.

O segundo processo tem como entrada um segmento do programa alvo, novamente representado pela forma intermediária, e uma hierarquia de programas primos. Ele tem como objetivo a determinação da função executada pelo programa alvo. Para alcançar tal objetivo, o processo de abstração deve executar as seguintes funções (a função 2.1 foi determinada no Capítulo 3):

2.1) Geração da abstração funcional de cada tipo de primo e posterior composição funcional para obter a abstração do todo;

2.2) Expansão das abstrações para que o resultado obtido seja mais facilmente interpretável.

A função 2.1, por ser mais complexa, é subdividida em quatro subfunções, quais sejam:

2.1.a) Verificar se a composição funcional entre duas condições é factível;

2.1.b) Determinar se um primo, que represente um "if" de predicado simples, pode ser abstraído numa forma mais concisa, que capture exatamente a mesma funcionalidade;

2.1.c) Resolver relações de recorrência em iterações, visando descrever o efeito da iteração sobre os dados que se encontram em seu corpo;

2.1.d) Permitir a intervenção do usuário nas situações em que seja necessária. O momento e a maneira como ocorre esta intervenção é melhor detalhada no transcorrer deste capítulo.

A Figura 4.2 descreve as funções que representam os subsistemas da FAF. Nesta figura, os retângulos com molduras duplas representam os subsistemas da FAF, os retângulos cheios indicam os módulos executados nos subsistemas, os círculos ilustram as saídas produzidas pelos módulos, as linhas cheias indicam fluxo de informação e, finalmente, as linhas tracejadas significam fluxo de controle.

Vale observar que a FAF é constituída por cinco subsistemas[1] que se comunicam por intermédio de arquivos. Um resumo da funcionalidade desses subsistemas que integram a Ferramenta de Abstração se faz necessário para melhor compreensão:

O subsistema "Interface de Abstração" é responsável pela comunicação do usuário com a FAF, indicando opções e critério de execução, o qual consiste na definição de quais variáveis e comandos serão analisados.

O subsistema "Gerador da Forma Intermediária" foi elaborado para realizar uma análise no código fonte do programa alvo, traduzindo-o para uma forma intermediária, que caracteriza o fluxo de controle e dados do programa original. Além disto, este subsistema gera uma tabela de símbolos e um conjunto de equações que representam os comandos (vide Apêndice E).

---

[1] O termo subsistema foi escolhido para designar as macro-funções da Ferramenta de Abstração e o termo módulo para designar os processos executados nos subsistemas. A utilização do termo subsistema é proposital e indica que os vários subsistemas da FAF podem ser implementados isoladamente ou em conjunto.

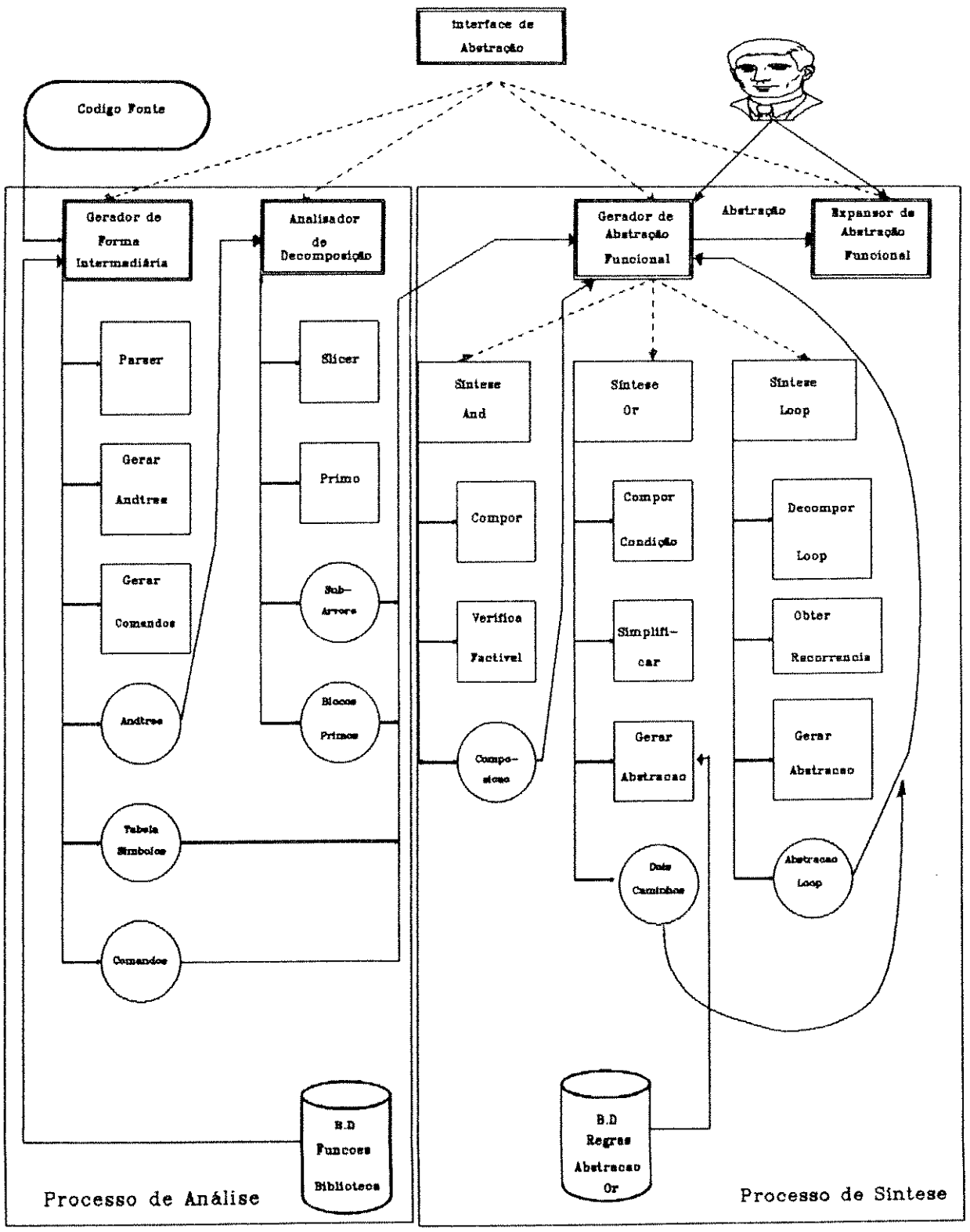


Figura 4.2: Arquitetura da Ferramenta de Abstração

O subsistema "Análise de Decomposição" é responsável pela subdivisão do programa de acordo com algum critério fornecido pelo usuário, utilizando-se a técnica de "Slicer" (Capítulo 2). A importância da subdivisão em questão reside no fato de que "slices" são geralmente menores do que o programa que os originou, sendo que reproduzem com fidelidade o comportamento do programa original em relação ao critério dado como entrada. Isto permite uma grande flexibilidade da análise do programa, isto é, pode-se analisar uma ou mais variáveis em relação ao programa inteiro ou a parte dele.

O subsistema "Análise de Decomposição" é responsável, também, pela obtenção da hierarquia de primos que compõem o programa, sendo que a essência da Ferramenta de Abstração Funcional se manifesta na análise dos mesmos.

O subsistema "Gerador da Abstração Funcional" é o principal subsistema da ferramenta. Ele é responsável pela análise estática de cada primo, pela composição funcional entre eles, pela análise de recorrência em iterações, pela simplificação e verificação da factibilidade de decisões e, ademais, possui interfaces que permitem ao usuário interferir em processos não totalmente automatizados.

Finalmente, o subsistema "Expansor da Abstração Funcional" foi desenvolvido para expandir as abstrações produzidas pelo subsistema "Gerador da Abstração Funcional", objetivando tornar o resultado obtido mais fácil de ser interpretado pelo usuário.

Portanto, a idéia da FAF é manipular um código fonte nos subsistemas descritos acima com o objetivo de derivar uma abstração funcional do programa alvo. O primeiro passo é produzir uma forma intermediária que represente todas as informações relevantes para a abstração; o segundo passo é segmentar o programa em função das variáveis requeridas pelo usuário; o terceiro, é proceder à abstração do programa em termos de seus programas primos, visando determinar o que acontece com as variáveis do

programa em todas as situações possíveis; e, por fim, o quarto passo é expandir o resultado obtido para melhor compreensão.

A linguagem alvo da FAF e o requisito básico desta ferramenta são enfatizados nas subseções 4.1.1 e 4.1.2.

4.1.1 Linguagem Alvo

A FAF irá analisar e abstrair programas escritos na linguagem de programação "C", a qual foi escolhida por ser considerada de médio nível, já que combina elementos das linguagens de alto nível com a funcionalidade da linguagem "assembly". Desta forma, a linguagem "C" pode manipular bits, bytes e endereços, sendo simultaneamente uma linguagem estruturada(2), que permite separar e esconder do resto do programa todas as informações necessárias para a execução de uma determinada tarefa [SCHI-91].

Vale adicionar que outras linguagens semanticamente mais simples do que a "C", podem ser facilmente incorporadas à ferramenta. No Apêndice A é apresentada a especificação em YACC(3) para a sintaxe da linguagem "C" utilizada neste trabalho.

4.1.2 Estruturação

Como foi visto nos capítulos anteriores, o código a ser abstraído deve estar estruturado, possuindo apenas os contrutores para decisão, seqüência e iteração. Além disto, cada bloco do programa deve possuir apenas um ponto de entrada e um ponto de saída. Neste sentido, a FAF será aplicada sobre códigos produzidos por uma ferramenta para estruturação baseada na aplicação do teorema da estrutura sobre o código original [MOUR-92].

---

(2) A linguagem "C" não pode ser formalmente considerada uma linguagem estruturada em blocos, pois não permite que funções ou procedimentos sejam declarados dentro de outros procedimentos ou funções [SCHI-91].

(3) YACC é um utilitário genérico utilizado para descrever uma especificação de uma linguagem. A partir desta especificação, o YACC gera uma sub-rotina em "C" que reconhece e executa as ações pertinentes à especificação implementada [STEP-92].

Os comandos "if, if-then-else" e "do-until" são utilizados, respectivamente, para representar os construtores de decisão e de iteração. Vale salientar, contudo, que o comando "do-until" foi escolhido por garantir a sua execução pelo menos uma vez, simplificando, portanto, a sua abstração. Ademais, se ele for derivado de algum comando do tipo "while" ou do tipo "for", um "if" deverá controlar a sua execução. Pode-se observar que apenas o comando "if" produzirá desvios no fluxo de controle do programa, caso o comando "do-until" seja escolhido para representar iterações.

Cada um dos subsistemas da ferramenta de abstração será detalhado nas próximas seções. Este detalhamento dar-se-á em termos de algoritmos, estruturas utilizadas e escopo de atuação da ferramenta.

4.2 Subsistema "Gerador da Forma Intermediária"

A primeira tarefa executada pela Ferramenta de Abstração é a tradução do programa alvo para uma forma intermediária que represente o fluxo de controle, o fluxo de dados e os comandos do programa a serem analisados.

Neste contexto, faz-se necessária a obtenção de uma estrutura de fácil uso, que possa ser utilizada em todos os subsistemas da FAF. Cabe ressaltar, que ela deve estar fortemente vinculada com a árvore "And-Or" descrita no Capítulo 2.

Esta árvore é um tipo abstrato de dados que possui os seguintes tipos de nós: "Or-Node", "And-Node", "Call-Node", "Loop-Node" e "Nill-Node". O significado semântico de cada nó e seu conteúdo são expostos a seguir:

Or-Node: Nó no qual apenas um dos filhos é executado, dependendo de alguma condição. Representa os "if's" das linguagens de programação usuais.

Conteúdo:

. Forma intermediária para a equação que representa a condição.

. Filho da esquerda representa a cláusula "then".
. Filho da direita representa a cláusula "else".
. Variáveis referenciadas no comando.
. Variáveis definidas no comando.

And-Node: Nó no qual todos os filhos serão executados incondicionalmente e em seqüência. Representa um bloco de programa.

Conteúdo:

. Possui n filhos que representam os comandos logicamente conectados pelo bloco em questão.

Loop-Node: Nó no qual todos os filhos serão executados de 0 a n vezes. Representa o comando de repetição "do-until".

Conteúdo:

. Possui n filhos que representam os comandos logicamente conectados pelo bloco em questão.
. Forma intermediária para a equação que representa a condição de término da iteração.
. Variáveis definidas no comando.
. Variáveis referenciadas no comando.

Call-Node: Nó que representa uma chamada a uma sub-routina.

Conteúdo:

. Forma intermediária para o comando em questão.
. Variáveis definidas no comando.
. Variáveis referenciadas no comando.
. Conjunto de parâmetros atuais. Cada parâmetro possuirá um nome e um tipo associado à forma de passagem do argumento (valor e referência).

Nill-Node: Nó que não possui filhos. Representa um comando sequencial simples (comando de atribuição).

Conteúdo:

. Forma intermediária para o comando em questão.

. Variáveis definidas no comando.

. Variáveis referenciadas no comando.

Assim sendo, a árvore "And-Or" foi utilizada durante essa pesquisa para representar as informações relevantes consideradas no processo de abstração funcional.

A utilização deste tipo abstrato de dados (árvore "And-Or") para representar um programa possui algumas vantagens. Em primeiro lugar, ela facilita a navegação em sua estrutura, seja percorrendo-a em ambos os sentidos ou dentro de um simples bloco de programa. Em segundo lugar, o algoritmo utilizado para a técnica de "Slicer" baseia-se na estrutura desta árvore. Por fim, a decomposição do programa em primos é fácil de ser gerada a partir desta árvore, como é descrito na Seção 4.3.

A consideração de tais vantagens demonstra que a árvore "And-Or" é bastante adequada para ser utilizada na ferramenta em desenvolvimento. O Apêndice B fornece a especificação para o tipo abstrato árvore "And-Or".

O programa da Figura 4.3 foi extraído de [MUCH-81] e deve ser considerado como base para a maioria dos exemplos apresentados a partir desta seção.

```
    extern double distance, time; extern int error;

    void Dck(station, starship, thrust, velocity, deltat)
    double station, starship, thrust, velocity, deltat;
    {
 1  double gconst, gravity, constacc, curvel, nextvel;
 2
 3  if (station <= 0.0 || starship <= 0 || thrust <= 0 ||
        velocity <= 0 || deltat <= 0.0 || time <= 0.0 ||
        distance <= 0.0) {
 4      error = 1;
 5  }
 6  else {
 7       gconst = 6.67 * 10.e-11;
 8       gravity  = gconst * station * starship /(distance*distance);
 9       constacc = gravity - thrust / starship;
10       curvel = velocity;
11       nextvel = curvel + constacc * deltat;
12
13       do {
14            distance = distance - curvel * deltat;
15            curvel   = nextvel;
16            time     = time + deltat;
17            nextvel = curvel + constacc * deltat;
18       } while(nextvel > 0.0);
19       error = 0;
   }
```

Figura 4.3: Programa "Docking" [MUCH-81]

A Figura 4.4 ilustra a árvore "And-Or" que corresponde ao programa "Docking". Vale observar que o número dentro do círculo indica a linha de código que o nó está representando.

A utilização desta árvore "And-Or" está mais voltada para representar o fluxo de controle e de dados de um programa alvo, fazendo-se necessária, contudo, a definição de outras informações primordiais para que a FAF possa ser empregada com pleno êxito. Neste sentido, a forma intermediária deverá colecionar informações sobre o programa original como um todo e distribuí-las em seu fluxo sequencial de execução, ou seja, nos nós da árvore "And-Or". Cabe salientar que tais informações se referem basicamente às variáveis (escopos, tipos e valores), à representação interna do comando e à condição de execução de um determinado caminho do programa, as quais serão descritas nas próximas subseções.

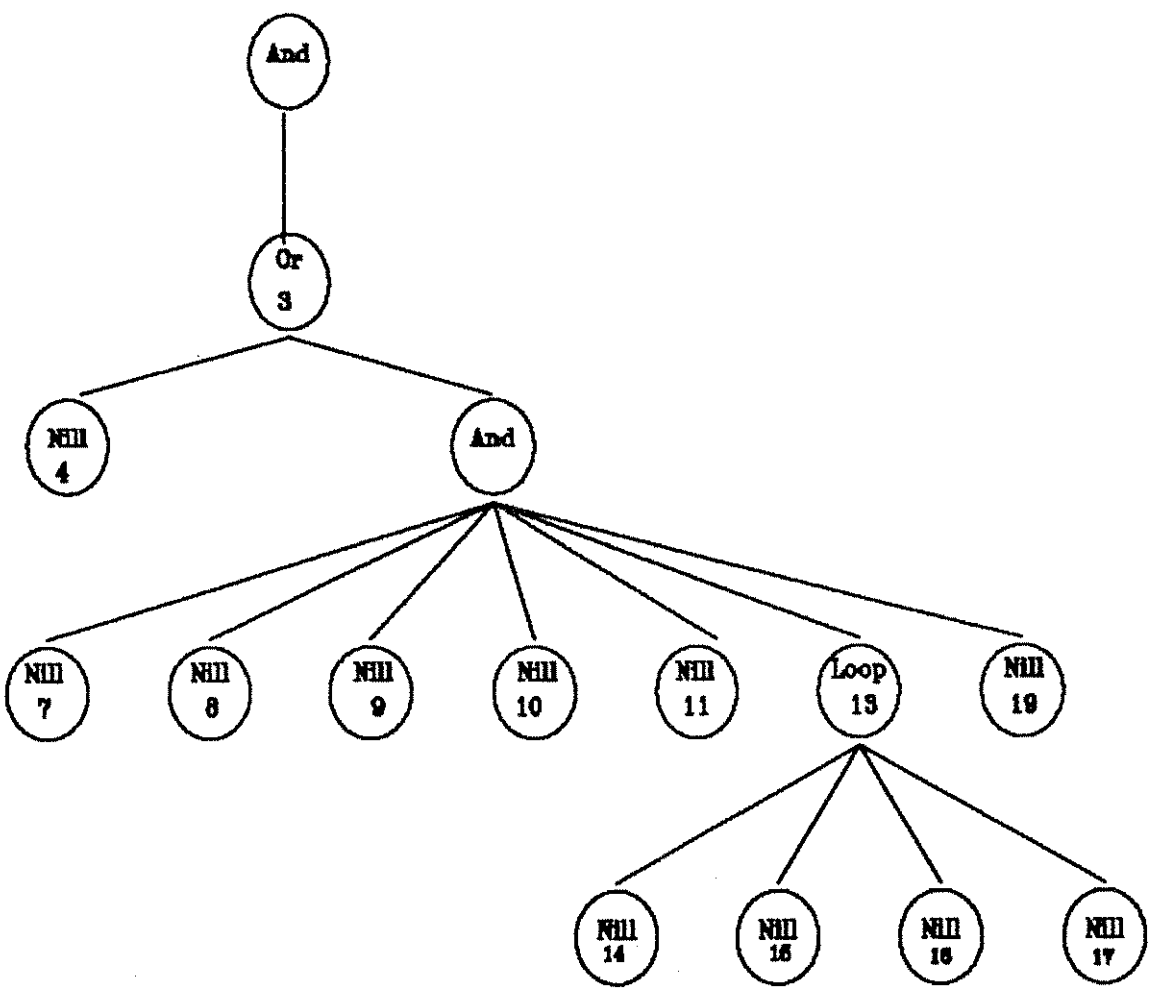


Figura 4.4: "And-Or-Tree" Correspondente ao Programa DOCKING.

4.2.1 Variáveis

Todo computador convencional fundamenta-se na noção de elementos celulares de memória que são identificados por um único endereço. O conteúdo de cada célula é o seu valor que pode ser lido ou modificado. Neste contexto, variáveis podem ser visualizadas como abstrações do conceito de células de memória e o comando de atribuição pode ser considerado como abstração do modificador de uma célula [WIRT-78].

Desta forma, todo processo de abstração está dirigido para linguagens de programação baseadas em atribuições e o seu resultado é guiado pelos valores assumidos pelas variáveis nos diversos caminhos de execução do programa. Neste sentido, é fundamental que os conceitos de escopo, de tipo e de valor da variável sejam incorporados à forma intermediária.

Escopo das Variáveis

O conceito de escopo de variáveis será mantido na forma intermediária obtida, sendo que o nome da variável será concatenado com o nome da função à qual ela pertence, seguido do escopo da variável, quando for o caso. Esta geração dos nomes das funções, seguidos do escopo das variáveis, será feita no momento da avaliação sintática do código fonte.

Por exemplo, uma variável "x" global terá o nome "x#g" na forma intermediária (observe que a letra "g" indica que a variável é global). Uma variável "y", local a uma função "z", terá o nome "y#z#1" na forma intermediária (observe que "y" é local à função cujo nome se encontra na expressão, no caso, "y" é local à função "z"). Por sua vez, uma variável (por exemplo, "y") declarada em um bloco interno à função "z", terá o nome "y#z#2" na forma intermediária (neste caso, além da variável "y" ser local à função "z", ela foi declarada em algum bloco de programa interno à "z", o que é indicado pelo número 2).

Tipo das Variáveis

O tipo de uma variável pode ser definido como uma especificação da classe de valores que podem ser atribuídos a uma variável. A determinação do tipo da variável será importante para o processo de padronização dos comandos condicionais (vide Capítulo 3) e, também, para determinar quais operações são legalmente utilizáveis para criar, acessar ou modificar o seu valor.

A forma intermediária deve armazenar as variáveis utilizadas e manter informações acerca dos seus atributos. Estes atributos são: 1) tipo básico (int, float, char, etc) e 2) indicador de "array", ponteiro, argumento ou função. Além do tipo básico, a variável arranjo ("array") deverá ser considerada e avaliada apropriadamente.

"Array"

A variável arranjo possui um índice que funciona como um cursor, o qual pode percorrer todos os seus elementos, sendo que a precisa posição do elemento apontado é muitas vezes não trivial ou mesmo indeterminável.

A forma intermediária deverá indicar em sua estrutura se uma determinada variável é arranjo e, sendo o caso, qual expressão que determina o elemento acessado. Todavia, as variáveis de tipo "array" não serão tratadas neste protótipo.

Valor das Variáveis

O valor que uma variável pode assumir está intimamente ligado ao seu tipo. No caso de uma variável de um programa escrito na linguagem "C", o seu valor pode ser numérico, alfabético ou, ainda, uma referência (ponteiro) para algum objeto, sendo somente passível de modificação via algum comando de atribuição. Cabe apenas ressaltar que a determinação da semântica dos comandos de atribuição requer a utilização de uma outra forma intermediária.

4.2.2 Forma Intermediária para os Comandos de Atribuição

O Capitulo 3 mostrou que as seqüências seriam analisadas pela técnica de "Trace-Table". A "Trace-Table" nada mais é do que uma tabela de equações cujas variáveis são substituídas pelas funções que as definem. Neste contexto, as equações de uma "Trace-Table" podem ser os comandos de

atribuição de um programa, os quais necessitam estar em uma forma intermediária para possibilitar a sua manutenção e manipulação pelo computador na aplicação da técnica de "Trace-Table".

Esta forma intermediária é, na verdade, uma estrutura de dados que representa um comando de atribuição em termos de expressões binárias, na qual cada linha corresponde a um operador e a dois operandos. Desta maneira, um comando pode ocupar várias linhas desta estrutura, chamada de dados abstratos do tipo tabela[4]. Por exemplo, a tabela de equações para o comando "distance = distance - currvel * deltat" é ilustrada abaixo:

	Operador	Operando1	Operando2
1	-	distance	EQ2
2	*	currvel	deltat

Esta tabela[5] corresponde a um grafo direto e acíclico de expressões binárias para um determinado comando do programa. A variável definida pelo comando apontará para a raiz deste grafo (primeira linha da tabela). Nesse tipo de representação, expressões e sub-expressões idênticas deverão apontar para a mesma tabela ou linha de uma tabela.

Entretanto, esta tabela não é suficiente para capturar toda a semântica que possa existir em um comando de atribuição em "C", o que torna necessário expandí-la em mais três colunas e, além disso, permitir que um de seus operandos possa ser, também, uma função. Sendo assim, a Tabela 4.1 descreve a especificação completa para o tipo abstrato tabela.

---

[4] Esta estrutura é comumente utilizada nas técnicas de execução simbólica de programas [MUCH-81].

[5] Observe que "Trace-Table" é uma técnica para abstração de seqüências e o tipo abstrato de dados tabela é uma estrutura definida para armazenar um comando de atribuição. As linhas de uma "Trace-Table" são constituídas de tabelas (expressões binárias) quando da implementação da FAF.

	Operador	Operando1	Operando2	Col4	Col5	Col6
1	+	Variável	Variável	-1	-1	-1
2	-	Tabela	Tabela	0	0	0
3	*	Constante	Constante	1	1	1
4	/	Função	Função	2	2	2
5	etc			3	3	3

Operador: + -> Soma
- -> Subtração
* -> Multiplicação
/ -> Divisão
^ -> Exponenciação
% -> Módulo de uma divisão
& -> And (Operador bit a bit)
| -> Or (Operador bit a bit)
<< -> Deslocamento à direita
>> -> Deslocamento à esquerda

Operando:

Variável: Indica uma variável.
Constante: Indica uma constante.
Tabela: Indica endereço de uma tabela (endereço em termos de linha e coluna), podendo ser a mesma tabela ou uma outra.
Função: Indica chamada a uma função. Esta função pode ser do programa, da biblioteca "C" ou uma abstração parcial.

Col4: Atua sobre o primeiro operando
Col5: Atua sobre o segundo operando
Col6: Atua sobre os dois operandos da linha

Neste caso, tem-se uma das seguintes possibilidades. Cada uma das quais atua sobre o primeiro, o segundo ou sobre os dois operandos, dependendo da coluna que estiver inserido :

0 --> Indica menos unário no operando em questão
1 --> Indica que o operando em questão é o conteúdo de um ponteiro (*Variável em 'C')
2 --> Indica que o operando em questão é um endereço
3 --> Indica a associação de menos unário e conteúdo de um ponteiro.
-1 --> Nada

Tabela 4.1: Especificação Completa do Conteúdo do Tipo Abstrato Tabela

A execução de algum comando do programa pode estar por vezes vinculada à execução de algum comando de decisão. Portanto, faz-se necessário definir uma forma intermediária para as expressões relacionais dos comandos de decisão.

4.2.3 Forma Intermediária para os Comandos de Decisão

A forma intermediária para os comandos de decisão baseia-se na estrutura definida para os comandos sequenciais e objetiva facilitar a manutenção dos operadores relacionais encontrados nos vários comandos de decisão do programa.

Como exemplo, o comando "IF (x <= 10 && y == 0 && z <= 20)" produziria a tabela da Figura 4.5.

	Operador	Operando1	Operando2
1	&&	EQ1	EQ2
2	&&	EQ3	EQ4
3	≤	x	10
4	=	y	0
5	≤	z	20

Figura 4.5: Tabela Original de um Comando Condicional

Entretanto, a tabela na forma acima deve ser padronizada para facilitar o processo de simplificação e verificação do comando de decisão. Esta padronização (vide Capítulo 3) possui três fases:

1. O booleano "NOT" é absorvido na expressão, utilizando-se o algoritmo descrito no Apêndice C.1.

2. Apenas os operandos "≤", "=" ou "≠" são utilizados. Ademais, a expressão à direita deve ser sintaticamente mais simples do que a expressão à esquerda, sendo que qualquer outra expressão é transformada para obedecer à forma padrão considerada. Nesta situação, são utilizadas as regras de tranformação listadas na Tabela 3.2.

3. As expressões lógicas são agregadas pelos booleanos "AND" ou "OR", sendo expressas na forma normal conjuntiva via o algoritmo do Apêndice C.2.

Após a aplicação das três fases descritas acima, é obtida uma tabela com todas as condições em forma normal conjuntiva, as quais são separadas em três conjuntos: igualdades (I), desigualdades (D) e disjunções (D).

Considere a Figura 4.6 que possui o resultado do processo de padronização aplicado à tabela da Figura 4.5.

	Operador	Operando1	Operando2
1	=	y	0
2	≤	x	10
3	≤	z	20

Figura 4.6: Representação em F.N.C da Expressão Lógica da Figura 4.5

Nesta tabela, cada linha representa uma condição, existindo um conector implícito ("AND") entre cada linha. A linha 1 representa o conjunto de igualdades; já as linhas 2 e 3 representam o conjunto de desigualdades.

A funcionalidade desta tabela deve ser aumentada através do acréscimo de mais três colunas para que ela possa capturar todas as peculiaridades da

linguagem "C" e, também, permitir a realização de transformações com o objetivo de se obter tabelas na forma normal conjuntiva, com os operadores lógicos ("≠" , "≤" e "=") e com a absorção do "NOT".

Considere o comando "if((a == 10) && (b ≥ 5)) && (!d || !e))", o qual possui todas as variáveis inteiras. A tabela gerada para este comando encontra-se na Figura 4.7.

	Operador	Operando1	Operando2	Col4	Col5	Col6
1	=	a	10	-1	-1	-1
2	≤	b	5	-1	3	-1
3	=	d	0	-1	-1	-1
4	=	e	0	-1	-1	-1

Figura 4.7: Tabela de Condições Estendida

Algumas explicações são necessárias para o completo entendimento desta tabela: a coluna quatro indica a utilização do "NOT" nos operandos e expressões da linha da tabela; a coluna cinco é utilizada para mascarar operandos que possuam menos unário ou indicador de endereço/conteúdo; e a coluna seis indica uma subtração de -1 ou -ε no segundo operando, no caso de necessidade de transformação dos operadores "<", ">" ou "≥" no operador "≤".

No exemplo dado, o número três da coluna cinco indica a aplicação do menos unário nos dois operandos, ou seja, transforma a expressão "b ≥ 5" em "-b ≤ -5". Observe, ainda, que esta tabela equivale à seguinte F.N.C:

And(Linha1, Linha2, Or(Linha3, Linha4)).

A semântica completa desta tabela aumentada é ilustrada na Tabela 4.2.

Operador	Operando1	Operando2	Col4	Col5	Col6
=	CONSTANTE	CONSTANTE	-1	-1	-1
≠	VARIAVEL	VARIAVEL	1	1	1
≤	FUNÇÃO	FUNÇÃO	2	2	2
	TABELA	TABELA	3	3	

Col4: Indica a presença do "NOT" nos operandos da linha
-1 -> Não existe NOT.
1 -> NOT no primeiro operando.
2 -> NOT no segundo operando.
3 -> NOT nos dois operandos, conjuntamente.

COL5: Indica a presença do menos unário nos operandos da linha. Neste caso, tem-se:
-1 -> Não existe menos unário.
1 -> Menos Unário no primeiro operando.
2 -> Menos Unário no segundo operando.
3 -> Menos Unário nos dois operandos, isoladamente.

11 -> Indica que o primeiro operando é um ponteiro.
12 -> Indica que o segundo operando é um ponteiro.
13 -> Indica que os dois operandos são ponteiros, isoladamente.

21 -> Indica Conteúdo do primeiro operando.
22 -> Indica Conteúdo do segundo operando.
23 -> Indica Conteúdo dos dois operandos, isoladamente.

31 -> Indica Menos Unário e Conteúdo do primeiro operando.
32 -> Indica Menos Unário e Conteúdo do segundo Operando.
33 -> Indica Menos Unário e Conteúdo dos dois operandos, isoladamente.

COL 6: Indica o uso de "-1" ou "-ε" nos operandos da linha
-1 -> Nada.
1 -> Subtrair (1) do segundo operando.
2 -> Subtrair (ε) do segundo operando.
ε -> Representa o menor real diferente de zero de um determinado computador.

Tabela 4.2: Semântica da Tabela de Condição

4.2.4 Função

A FAF trabalha com chamadas às funções desde que elas já tenham sido analisadas em relação às suas Variáveis de Estado. Isto exige uma restrição com relação à prioridade de avaliação das expressões que contenham chamadas às funções, ou seja, a função deve ser avaliada e executada prioritariamente. For exemplo, considere a expressão abaixo, na qual a variável "y" é global:

x = f(w) + y + g(h).

Não há garantia de que f(w) ou g(h) alterem ou não o valor da variável y e, dependendo da ordem de avaliação dos fatores, resultados diferentes podem ser obtidos. Para resolver este problema e reduzir os recursos necessários de máquina (memória e tempo de CPU), todas as funções que são chamadas dentro de uma função devem ser analisadas em primeiro lugar.

Além do exposto anteriormente, será considerada a passagem de parâmetros por referência e/ou valor. Em qualquer dos casos, a função chamadora herda o comportamento da função chamada, observando a troca dos nomes das variáveis (parâmetro formal pelo parâmetro atual) pelo método posicional. A linguagem intermediária mantém os parâmetros formais de uma função, bem como os parâmetros atuais de cada função chamada. Sendo assim, o primeiro nó "And" da árvore "And-Or" da função possuirá uma lista com o nome e tipo dos seus parâmetros formais e cada nó de tipo "call" (chamada de sub-rotina) possuirá uma lista com o nome e tipo dos parâmetros atuais.

A função pode ser de dois tipos do ponto de vista da FAF. O primeiro tipo engloba as funções implementadas pelo usuário. Neste caso, é necessário analisá-las em termos de "como é feito" para se derivar as regras de "o que é feito". O segundo tipo engloba as funções de biblioteca e as funções abstrações geradas internamente ou fornecidas como abstração pelo usuário (decisão e iteração), para as quais a preocupação residem "no que é feito". Neste sentido, um banco de funções integra a ferramenta para armazenar as abstrações ("o que é feito") das funções de biblioteca.

A Tabela 4.3 retrata a sintaxe utilizada para representar a funcionalidade destas funções.

```
Nome da Função ( Arg1, Arg2, ..., Argn)

Descrição {
      <Função : "Texto até 256 caracteres">
      <Arg1>  : <Tipo> <"Texto até 80 caracteres">
      <Arg2>  : <Tipo> <"Texto até 80 caracteres">
        .
        .
        .
      <Argn>  : <Tipo> <"Texto até 80 caracteres">
}

Especificação {
      <Especificação da Função>
}

CriarOperador {
      Mapeamento : Expressão
}
```

Tabela 4.3: Sintaxe para as Funções de Biblioteca(6)

Algumas explicações são necessárias para se entender esta tabela:

i) Atributos entre os caracteres "<" e ">" são opcionais.

ii) Apenas o nome da função e seus argumentos são obrigatórios.

iii) O comando especificação foi inserido para possibilitar uma futura evolução da ferramenta, permitindo que as funções venham a ser especificadas formalmente.

iv) O comando "CriarOperador" é utilizado apenas para as funções de biblioteca. Ele permite que algumas funções possam ser especificadas em termos de expressões mais simples de serem manipuladas e entendidas. Por exemplo, considere a função "STRCPY(Arg1,Arg2)" da biblioteca "C" (a qual copia o conteúdo de

(6) Esta sintaxe também será empregada para permitir que o usuário forneça abstrações parciais de iterações e decisões, contudo, sem a possibilidade de se entrar com o comando "CriarOperador".

Arg2 em Arg1), bem como a seguinte especificação do comando "CriarOperador":

```
CriarOperador {
     Função : Arg1 = Arg2
}
```

Esta especificação permite, por exemplo, que a função "STRCPY" seja substituída pela expressão "Arg1 = Arg2" na forma intermediária. Vale observar que esta substituição é executada em tempo de geração da árvore "And-Or".

Cabe salientar três situações que podem ocorrer quando uma função de biblioteca é encontrada e sua ocorrência é procurada no banco de funções:

iv.a) Não existe ocorrência:

O nome da função e seus argumentos deverão fazer parte da forma intermediária, sendo o caso das funções matemáticas Seno, Coseno, etc.

iv.b) Existe ocorrência e foi especificado o comando CriarOperador:

A função e seus argumentos serão substituídos pela expressão especificada no comando CriarOperador. A sintaxe completa deste comando será:

Mapeamento	**Substituição**
Função	Expressão*
ou	
Função OpRelacional Constante	Expressão*

* Expressão:

- é atuante apenas sobre os argumentos.
- pode ser qualquer expressão válida em "C".
- pode ser uma descrição textual concatenada com argumentos via o símbolo "#", isto é: "Texto"# Arg_i # "Texto" # Arg_i...

Considere as seguintes funções de biblioteca relacionadas com cadeias de caracteres:

Função	Mapeamento	Substituição
STRCPY(A, B)	FUNÇÃO	Arg1 = Arg2
STRCMP(A, B) == 0	FUNÇÃO == 0	Arg1 == Arg2
STRCMP(A, B) <= 0	FUNÇÃO <= 0	Arg1 <= Arg2

Note que a idéia é sempre obter abstrações mais simples, devendo o comando "CriarOperador" ser utilizado parcimoniosamente.

iv.c) Existe ocorrência, porém sem o comando CriarOperador

A função foi comentada com o comando "Descrição". Este comentário fará parte da função e poderá ser utilizado nas abstrações parciais (interface com usuário) e, ainda, na abstração final.

4.3 Subsistema "Análise de Decomposição"

A estratégia "dividir para conquistar" é uma das principais diferenças que este modelo de abstração proposto possui em relação às técnicas tradicionais de execução simbólica. Em primeiro lugar, permite que o programa alvo seja abstraído em termos mais próximos da sua funcionalidade desejada, ou seja, em relação às suas Variáveis de Estado. Em segundo lugar, admite que pequenos blocos de programa possam ser sintetizados em mais alto nível (vide Seção 4.4). Estes dois fatores garantem uma maior flexibilidade e, muitas vezes, produzem resultados mais concisos.

O subsistema de decomposição é formado por dois módulos: módulo para segmentação via utilização do algoritmo da técnica de "Slicer" e módulo para decomposição em programas primos.

Módulo para Segmentação via "Slicer"

Este módulo permite que a estratégia "dividir para conquistar" possa ser aplicada à duas situações. A primeira se refere à possibilidade do usuário especificar as variáveis de seu interesse e as linhas do programa que devem ser abstraídas. Neste caso, o usuário deve fornecer um conjunto SC (critério para o "Slicer"), tal que SC=(V, P, n1, n2), indicando que se devem extrair os comandos de P, entre as linhas n1 e n2, que afetam direta ou indiretamente as variáveis do conjunto V.

O algoritmo descrito no Apêndice C.3 é empregado para se obter esta segmentação. Este algoritmo tem como entrada uma quadrupla SC e uma árvore "And-Or" correspondente ao programa P, além de gerar como saída uma subárvore com os nós que afetam V.

A segunda situação prevê que o "Slicer" possa ser invocado pelo subsistema "Gerador de Abstração Funcional" para que ele determine todas as computações executadas por uma determinada iteração. Esta situação é melhor detalhada na Seção 4.4.

Módulo Para Decomposição em Programas Primos

A subárvore gerada anteriormente será a entrada no módulo de decomposição em primos. Esta decomposição é alcançada por meio dos algoritmos listados nos Apêndices C.4 e C.5.

O módulo recebe uma subárvore e produz uma lista de nós que lideram outros nós. Cada líder possui um tipo ("And", "Or" ou "Loop") e sua posição na lista indica a ordem na qual os blocos devem ser analisados.

A Figura 4.8 apresenta de forma diagramática o subsistema "Análise de Decomposição".

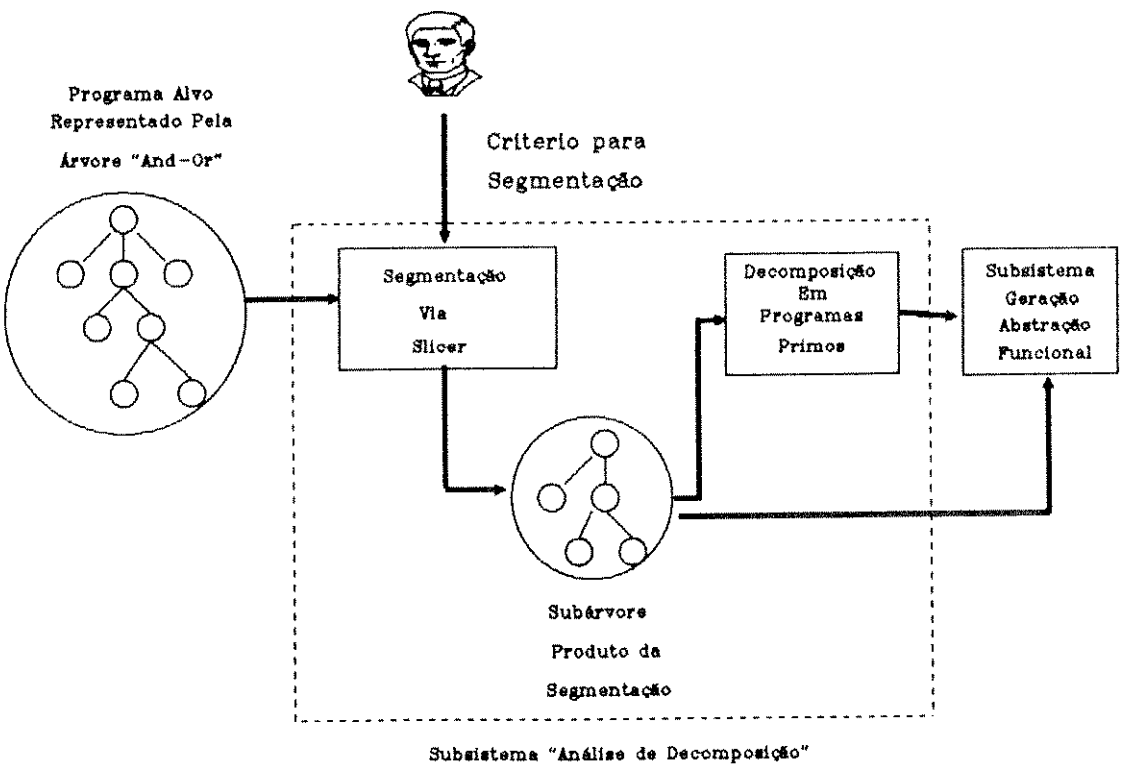


Figura 4.8: Subsistema "Análise de Decomposição"

4.4 Subsistema "Geração de Abstração Funcional"

O subsistema "Geração de Abstração Funcional" recebe como entrada um programa ou parte dele, representados pela árvore "And-Or", e um conjunto de nós que correspondem aos primos deste programa, produzindo como resultado uma descrição do efeito do programa alvo sobre os seus dados. Esta descrição é livre das estruturas de controle tradicionais (iteração, decisão ou seqüência) e é obtida gradativamente pela composição dos primos do programa.

Cada programa primo é representado a partir de três tipos básicos detalhados a seguir:

i) Caso : Todo resultado obtido é expresso em termos de possíveis casos executáveis pelo programa. Cada caso possui uma condição que delimita como ele pode ser sensibilizado e, além disto, aponta para um conjunto de atribuições concorrentes.

ii) Atribuições Concorrentes:

A atribuição concorrente é definida como atribuição simultânea, traduzindo o que um determinado caso executa. O valor de uma variável é considerado como o efeito do programa primo sobre ela. Este valor é uma expressão simbólica representada por uma tabela (vide Seção 4.2.2). O conteúdo desta tabela pode ser, por sua vez, tabelas, funções, constantes ou variáveis.

iii) Função:

Toda abstração de mais alto nível fornecida pelo usuário, assim como as funções pertencentes ao banco de funções de biblioteca e as funções abstrações obtidas sobre o corpo de iterações (cuja sintaxe é fornecida pela Seção 4.2.4) são consideradas como pertencentes a este tipo básico de representação denominado "função".

A Figura 4.9 ilustra a estrutura de dados abstrata do tipo "caso". Esta figura mostra que cada primo poderá conter "n" casos. Cada caso possui uma determinada condição de execução (CE) na forma normal conjuntiva e um conjunto de variáveis definidas, que apontam para tabelas.

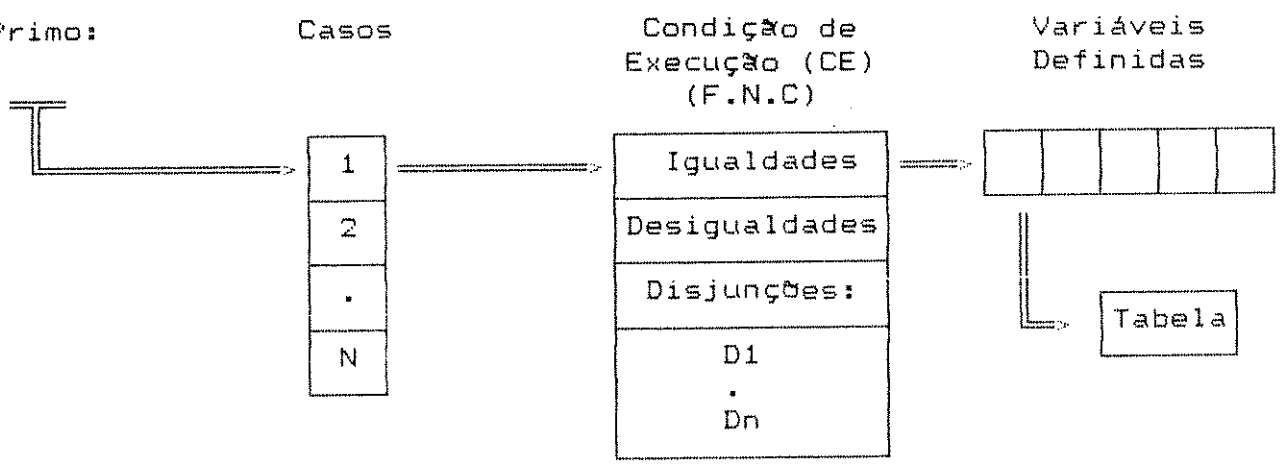


Figura 4.9: Tipo Abstrato Caso

A análise e composição funcional de cada primo dar-se-á por meio do Algoritmo 4.1 especificado a seguir:

Algoritmo 4.1: Síntese/Composição de primos

Entrada : A : Arvore "And-Or".
BL : Hierarquia de primos.

Saída : O primeiro nó "And" de A possuirá "N" casos, indicando as possíveis computações do programa.

Descrição : O método é aplicado a cada elemento de "BL", sendo este elemento um nó de A. Quando um primo de mais alto nível for sintetizado, todos os primos de mais baixo nível já terão sido abstraídos devido à construção de "BL".

Referência : Elaborado pelo autor da pesquisa.

Método :

1. Para cada elemento **i** (que representa um nó) de "**BL**" :

    1.1 Se o tipo do nó **i** é "**And**"

        . Aplicar ao nó o **Algoritmo 4.2** (vide Seção 4.4.1)

    1.2 Se o tipo do nó **i** é "**OR**"

        . Aplicar ao nó o **Algoritmo 4.4** (vide Seção 4.4.2)

    1.3 Se o tipo do nó **i** é "**LOOP**"

        . Aplicar ao nó o **Algoritmo 4.6** (vide Seção 4.4.3)

Este algoritmo obtém a abstração funcional do programa P via análise de cada primo (seqüência, decisão ou iteração) de forma distinta. Nas três próximas subseções são detalhados os procedimentos necessários para se obter a abstração de cada tipo de primo.

4.4.1 Síntese de Seqüência

Um traço (composição) do último ao primeiro filho do nó "AND" em questão deve ser executado com o objetivo de abstrair a função programa de seqüências. Neste traço, cada variável do lado direito de uma expressão será substituída pela expressão simbólica que a define, se existir um comando anterior que defina esta variável.

Este processo será repetido, recursivamente, com os Algoritmos 4.2 e 4.3 para todas as combinações dos casos envolvidos nos nós, segundo a técnica de "Trace-Table". Por exemplo, suponha que se esteja analisando um nó "AND" com três filhos e cada filho contém dois possíveis casos de execução. A análise completa deste primo irá gerar uma abstração com oito possíveis casos, se todas as composições forem factíveis.

A tarefa de verificação e simplificação das condições envolvidas na composição entre dois casos proceder-se-á via o Algoritmo 4.5. No entanto,

se este algoritmo não puder determinar a infactibilidade da conjunção das condições, o usuário será o responsável por esta verificação.

Um novo caso com "N" possibilidades será gerado e armazenado no nó "And" em questão, quando o processo estiver concluído.

Algoritmo 4.2: Síntese de seqüências.

Entrada : Pai : Nó "And" a ser sintetizado.

Saída : Abstração com "N" Casos.

Descrição : Este algoritmo utiliza o Algoritmo 4.3, recursivamente, com o intuito de se obter a composição funcional de todos os casos envolvidos nos filhos do nó pai.

Referência : Elaborado pelo autor da pesquisa.

Método :

1. Seja **Filhod** o filho mais à direita do **Pai.**

2. Para cada **casoi** de **Filhod:**

    2.1 Se existir irmão à esquerda do **Filhod:**

        2.1.1 Seja **Filhoe** irmão imediatamente à esquerda de **Filhod.**

        2.1.2 Chamar o **Algoritmo 4.3**, passando **Casoi** e **Filhoe.**

    2.2 Se não existir irmão à esquerda do **Filhod**

        2.2.1 Inserir **Casoi** em **Pai.**

Algoritmo 4.3: Composição Funcional

Entrada : Casod e Filho.

Saída : Caso, resultante da composição de Casod e Filho.

Descrição : Será verificado se existe alguma variável referenciada em Casod que tenha sido definida em Filho. Se existir, deve-se substituir a variável pela expressão simbólica que a define.

Referência : Elaborado pelo autor da pesquisa.

Método :

1. Para cada **Casoj** de filho:

   1.1 Se a condição de execução (CE) de **Casod** puder ser composta funcionalmente com a CE de **Casoj**, utilizando-se o **Algoritmo 4.5**, então:

      1.1.1 Para cada variável referenciada em **Casod**, verificar se existe alguma expressão em **Casoj** que a defina. Caso exista, trocar variável referenciada pela expressão simbólica que a define, gerando um novo caso: **Casoz.**

      1.1.2 Se existe irmão à esquerda de **Filho:**

         1.1.2.1 Faça **Filhoe** irmão imediatamente à esquerda de **Filho.**

         1.1.2.2 Chamar recursivamente o **Algoritmo 4.3**, passando **Casoz** e **Filhoe.**

      1.1.3 Caso contrário, inserir **Casoz** em **Pai.**

4.4.2 Síntese de Decisões

A síntese de decisões requer, em primeiro lugar, a verificação da possibilidade de abstração em mais alto nível do nó "OR" e de seus filhos. Em segundo lugar, caso o nó não possa ser abstraído em mais alto nível, deve-se quebrar o processo em dois casos - a parte "Then" e a parte "Else"- e continuar o processo de composição em cada parte, separadamente, para depois combiná-las. Se, por exemplo, a parte "Then" possuir "r" casos e a "Else", "s" casos, a abstração combinada possuirá "r+s" casos. Neste momento, deve ser verificado novamente se as composições das condições são factíveis para evitar uma explosão de casos.

O Algoritmo 4.4 a seguir retrata os passos necessários para abstração de decisões:

Algoritmo 4.4: Síntese de Decisões

Entrada : Pai : Nó "Or" a ser analisado.

Saída : Análise do nó "Or".

Descrição : Será gerado "N" casos, obtidos da soma dos casos do filho "Then" com os casos do filho "Else".

Referência : Elaborado pelo autor da pesquisa.

Método :

1. Pedir ao usuário uma abstração de mais alto nível para o primo analisado.

   1.1 Se o usuário fornecer abstração:

      1.1.1 armazená-la no nó **PAI** e voltar para o **Algoritmo 4.1.**

2. Se o usuário não fornecer a abstração:

   2.1 Para cada **Casoi** do filho da parte "Then":

      2.1.1 Compor a CE de **Pai** com a CE do **Casoi.**

      2.1.2 Verificar/simplificar a composição obtida em 2.1.1, segundo **Algoritmo 4.5.**

      2.1.3 Se o resultado do Passo 2.1.2 for factível, gerar novo **Caso** em **Pai.**

   2.2 Repetir Passos 2.1.1 a 2.1.3 para cada **Casoj** do filho referente à parte "Else".

Algoritmo 4.5: Simplificação/Verificação de Condições

Entrada : C : Conjunção atual.
B : Nova Conjunção.

Saída : 1. Um novo conjunto formado pela conjunção de C e B.
2. Um conjunto vazio, caso a conjunção seja considerada não factível.

Descrição : Comparar cada conjunção de C com as conjunções de B, procedendo-se às simplificações e verificando se estas são factíveis. Este algoritmo tem como base o algoritmo descrito na Seção 3.3.2.1.

Notação : Considere C e B da forma:
And(Ii,..., Ie, Oi,..., Oo, Di, ..., Dd). Onde:
Ii -> Conjunto de Igualdades.
Oi -> Conjunto de Outras unidades relacionais.
Di -> Conjunto das Disjunções.

Referência : **[CHEA-79]**

Método :

1. Para cada Conjunção **Bj** de B, execute os Passos **1.1** até **1.4:**

1.1 Operações com as igualdades **I1, ..., Ie** de **C.**

. Estas igualdades são substituídas apropriadamente em **Bj**, isto é, para cada **Ii** que possui a forma "**Lj = Rj**", troca-se cada ocorrência de **Lj** em **Bj** (se existir) por **Rj**. Neste caso, três situações podem ocorrer:

i) **Bj** é reduzida para a cláusula verdadeira:
. **Bj** pode ser ignorada.

ii) **Bj** é reduzida para a cláusula falsa:
. A conjunção de **C** com **B** é considerada infactível.

iii) São incomparáveis:
. Não se pode decidir sobre a factibilidade das conjunções.

1.2 Se **Bj** é uma igualdade:

1.2.1 Para cada conjunção de **C** que possua a forma "Lj operador Rj":

. Proceder à substituição de **Lj** por **R** de **Bj**, se **Lj** igual a **L** de **Bj**. Novamente, deve-se eliminar qualquer conjunção cuja substituição seja reduzida para o booleano verdadeiro. Se a substituição for reduzida para falso, deve-se considerar infactível a conjunção de **C** com **B.**

. Inserir **Bj** no conjunto das igualdades de **C.**

1.3 Se **Bj** é uma outra unidade relacional:

1.3.1 **Bj** é conjugado com cada unidade relacional **(O1,...,On)** e com cada igualdade de **C.** Desta forma, vários resultados podem ocorrer:

i) O resultado da conjunção é falso e toda a conjunção deve ser eliminada, por exemplo:

. Oi = (L ≤ R) e B = (-L ≤ -R-1)
. Oi = (-L ≤ -R) e B = (L ≤ R-1)
. Oi = (L = R) e B = (L ≠ R)
. Oi = (L = R) e B = (L ≤ R-1)
. Oi = (L = R) e B = (-L ≤ -R-1)
. Etc

ii) O resultado da conjunção de **Bj** com **Oi** pode produzir uma outra relação de desigualdade. Por exemplo:

. Oi = (L ≤ R) e Bj = (L ≠ R) ---> L ≤ R-1
(se R e L são inteiros)
. Oi = (L ≤ 10) e Bj = (L ≤ 0) ---> L ≤ 0
. Oi = (L ≤ 10) e Bj = (-L = -10)---> L = 10
. Oi = (L ≤ R) e Bj = (R ≤ C) ---> L ≤ C
. Oi = (L ≤ R) e Bj = (-L ≤ -R) ---> L = R
. Etc

Em todos estes casos, **Oi** e **Bj** são eliminados e a nova relação produzida é adicionada à **C**.

iii) Os dois são incomparáveis, então, **Bj** deve ser inserido no conjunto Outras unidades relacionais de **C**.

1.3.2 **Bj** é analisado contra cada disjunção (**Di, ..., Dn**), aplicando os Passos **1.1, 1.2 e 1.3.1.** Podem ocorrer simplificações, por exemplo:

. **Bj** = (L ≤ R)

Di = Or((-L ≤ -R), D2,..., Dn)
Resolvendo-se **Bj** e **Di**, tem-se:
Or((L = R), D2,..., Dn), que deve substituir **Di**

1.4 Se **Bj** é uma disjunção:

. Simplificar **Bj** em relação às conjunções de **C** (conjuntos **I** e O). Podendo ocorrer:

i) Existe algum **Ii** ou **Oi** que seja uma condição equivalente à ou mais forte que alguma condição de **Bj**:

. Eliminar a condição de **Bj**. Por exemplo:

. **Bj** = Or((L ≤ R), A ≠ 10)
  Oi = (L ≤ R-1)

  Resolvendo-se L ≤ R de **Bj** contra **Oi**, tem-se:

  Bj = (A ≠ 10), **Bj** foi reduzido para uma conjunção.

ii) Existe **Ii** ou **Oi** que seja uma condição contraditória com relação à alguma condição de **Bj**:

. Eliminar a condição de **Bj**. Por exemplo:

. **Bj** = Or((L ≤ R), A ≠ 10)
  Ii = (L = R)

  Resolvendo-se L ≤ R de **Bj** contra **Ii**, tem-se:

  Bj = (A ≠ 10), **Bj** foi reduzido para uma conjunção.

. Se todas as condições de **Bj** forem eliminadas por contradição, deve-se considerar a conjunção de **C** com **B** infactível.

. Se restar mais do que uma condição em **Bj**, este deve ser inserido no conjunto **D** de **C**.

. Se restar apenas uma condição em **Bj**, este foi reduzido para uma conjunção, devendo ser inserido no conjunto **I** ou **O** de **C**.

4.4.3 Síntese de Iterações

A FAF possui recursos para determinar o efeito, em alguns casos, da iteração sobre as variáveis definidas em seu corpo, permitindo que se faça uma análise global do programa, ao invés de se percorrer um determinado caminho como nas técnicas usuais de execução simbólica.

A técnica adotada para derivar abstrações de iterações foi apresentada no Capítulo 3. Resumidamente, a técnica de síntese baseia-se na tentativa de desenvolvimento das relações de recorrência existentes no corpo da iteração. Porém, a resolução automática das relações de recorrência não é trivial, principalmente no que diz respeito às relações que envolvem muitas variáveis dependentes umas das outras e às relações condicionais. Desta forma, a iteração é segmentada em função de seus elementos computacionais

básicos (vide Capítulo 2), ou seja, computações independentes, filtros e "loop" básicos, objetivando facilitar a resolução das relações de recorrência.

O algoritmo de análise e síntese aqui apresentado baseia-se nesta segmentação e na tentativa de se obter uma expressão que determine o número de ciclos executados pela iteração. Por outro lado, qualquer iteração que não possa ser sintetizada completamente por este algoritmo deverá ter sua funcionalidade fornecida pelo usuário.

Pode-se apontar cinco passos genéricos para cada variável X, cujo valor possa ser modificado durante um determinado ciclo [CHEA-79]:

i) Seja X_k o valor de indução, representando o valor de X no início da k-ésima iteração.

ii) Determinar o valor de cada variável X, assumindo-se um novo ciclo para o "loop". Este valor é denotado por X_{k+1} e equivale ao novo valor de X.

iii) Sejam P1, P2, ..., Pn os predicados que determinam as possíveis condições de término do "loop". Considere, ainda, E1, E2, ..., En como expressões que alteram as variáveis contidas nos predicados.

iv) Determinar o número de ciclos do "loop" (IL) em função dos pares: (P1, E1), (P2, E2), ..., (Pn, En).

v) Desenvolver a relação de recorrência em função de IL para cada par (X_k, X_{k+1}). Esta solução, denotada X(k), representa o valor simbólico da variável X no início do k-ésimo ciclo.

Os passos 1 a 5 devem ser particularizados para as classes de relações de recorrência que esta ferramenta suportará.

Relações de Recorrência Aritmética

As relações de recorrência aritmética podem ser sintetizadas automaticamente em um grande número de casos. Porém, o Algoritmo 4.6 foi idealizado para trabalhar apenas com as relações invariantes, soma/subtração e produto/divisão, que podem ser, por sua vez, univariadas, simultâneas ou condicionais.

Algoritmo 4.6: Síntese das Relações de Recorrência Aritmética

Entrada : L : "Loop"

Saída : Abstração das variáveis definidas em L.

Descrição : O algoritmo será dividido em três partes. A primeira tem a função de segmentar L em termos de suas computações, "loop" básico e filtros. A segunda analisa cada segmento na tentativa de resolver as relações. A terceira visa determinar o número de ciclos que o "loop" irá executar.

Notação : X(k) : Solução da relação de recorrência de X para o k-ésimo ciclo.

X_K : Valor de indução, que representa o valor de X no k-ésimo ciclo.

$X_{K+1} = F(X_K, Y_K, k)$: Equação que representa um relação de recorrência simultânea, pois X_{K+1} depende de X_K, de Y_K e do ciclo K.

$Y_{K+1} = G(Y_K, k)$: Relação de recorrência univariada.

Referência : Elaborado pelo autor da pesquisa a partir das relações de recorrência estudadas por Cheatham [CHEA-79].

Método :

Parte 1 : Segmentação

1.1 Seja V_C = { V1, ..., Vn} o conjunto de variáveis que recebem valor dentro do corpo de L.

2.2 Seja V_P = { V1, ..., Vm} o conjunto de variáveis referenciadas nos predicados que determinam as condições de parada de L.

1.3 Seja S_C = { S1, ..., Sn} um conjunto formado por "Slices". Cada "Slice" (Si) corresponde aos comandos do corpo de **L** que afetam o comportamento da variável **Vi** do conjunto V_C.

1.4 Seja S_P = { S1, ..., Sm} um conjunto formado por "Slices". Cada "Slice" (Si) corresponde aos comandos do corpo de **L** que afetam o comportamento da variável **Vi** do conjunto V_P.

1.5 Extrair os filtros de cada "Slice" de S_C.

1.6 Extrair os filtros de cada "Slice" de S_P.

Parte 2: Resolução da Relações de Recorrência

2.1 Fazer **S** = S_P U S_C

2.2 Determinar a ordem de avaliação de **S**, de tal modo que:

- Em primeiro lugar, estejam as relações invariantes.
- Em segundo lugar, as relações de recorrência que possam ser resolvidas diretamente, ou seja, da forma: $Y_{K+1} = G(Y_K, K)$.
- Em terceiro lugar, as relações de recorrência que dependam de apenas uma outra variável, isto é, $X_{K+1} = F(X_K, Y_K, K)$.
- Em quarto lugar, as relações que dependam de outras duas variáveis: Xk+1 = $F(X_K, Y_K, Z_K, k)$.
- Continuar assim, sucessivamente, até que todos os "Slices" de **S** estejam ordenados.

2.3 Para cada "Slice" de **S**, desenvolver as relações de recorrência existentes dentro dele de acordo com os seguintes passos:

2.3.1 Se a relação é invariante:

- A solução é $X(K) = X_1$, isto é, o valor de X em qualquer ciclo é igual ao seu valor inicial.

2.3.2 Se a relação é univariada:

- Executar Passos **2.4** e **2.5**

2.3.3 Se a relação é simultânea:

. Substituir a solução obtida para as variáveis independentes da relação, reduzindo o número de relações simultâneas. Por exemplo:

$X_{k+1} = F(X_k, Y_k, k)$. Dado que I_k já foi analisado. Então, $X_{k+1} = F(X_k, Y(k), k) = F1(X_k, k)$

. Executar passos **2.4.1** até **2.4.5** e **2.5** para cada nova relação obtida.

2.4 Resolução das relações de recorrência direta:

2.4.1 Se a relação é invariante:

. A solução é: **$X(k) = X_1$**

2.4.2 Se a relação é uma soma da forma: $X_{k+1} = X_k + G(k)$

. A solução é: **$X(k) = X_1$ + Soma_Finita(j, 1, k-1, G(j))**

2.4.3 Se a relação é uma subtração da forma: $X_{k+1} = X_k - G(k)$

. A solução é: **$X(k) = X_1$ - Soma_Finita(j, 1, k-1, G(j))**

2.4.4 Se a relação corresponde a uma multiplicação da forma: $X_{k+1} = X_k * G(k)$

. A solução é: **$X(k) = X_1$ * Produto_Finito(j, 1, K-1, G(j))**

2.4.5 Se a relação corresponde a uma divisão da forma: $X_{k+1} = X_k / G(k)$

. A solução é: **$X(k) = X_1$ / Produto_Finito(j, 1, K-1, G(j))**

2.5 Se existir um filtro associado à qualquer relação de recorrência analisada nos passos anteriores:

. Associar um "Case" à solução encontrada com o seguinte formato: **CASE(predicado, ação).** Por exemplo, considere o seguinte bloco de programma:

```
do {
   .
   if (x > 10)
       y = y + F(K)
   .
} while();
```

A solução seria :

$$y(k) = y_1 + \text{Soma\_Finita}(j,1,K-1,\text{Case}(x>10,F(K)))$$

Observe que x na expressão acima poderia ser uma outra relação de recorrência.

Parte 3: Determinação do número de ciclos de L

Descrição : A idéia básica é associar as condições com os comandos que podem alterar o seu comportamento. Os elementos básicos analisados são:

i) Valor inicial da progressão

ii) Termo da progressão

iii) Operador condicional utilizado.

Observe que as condições envolvidas estão na forma normal conjuntiva (And (I1,..., In, O1, ..., Om, (D11, ..., D1i), ..., (Dj1, ..., Djl)) e que apenas os operadores "≤", "=" e "≠" são utilizados.

Método:

3.1 Para cada "Slice" (**Si**) de S_P:

3.1.1 Analisar separadamente cada expressão de igualdade (**Ii**), de desigualdade (**Oi**) ou disjunção (**Dij**):

. Recuperar a condição associada ao "Slice" (**Si**)

. Recuperar o comando que incrementa as variáveis da condição.

3.1.1.1 Verificar se a condição e expressão de incremento são passíveis de análise, ou seja:

. Condição : X operador1 expr2

. Incremento: X = X operador2 expr3

3.1.1.2 Considere:

. X : Variável incrementada
. expr2 : Termo independente da condição
. expr3 : Termo que incrementa a variável X

3.1.1.3 Se o operador de comparação é "≤" e a expressão incremento é uma progressão aritmética, tem-se:

N = (1 + Floor ((expr2-X) / expr3))

3.1.1.4 Se operador de comparação é "≠" e a expressão incremento é uma progressão aritmética, tem-se:

N = (Ceiling(expr2-X) / expr3)

3.1.1.5 Se a expressão incremento é uma progressão geométrica:

. Reduzir a expressão para soma com uso de logaritmo e, então, aplicar os passos **3.1.1.3** ou **3.1.1.4.**

3.1.2 Se não for possível analisar, será requerida a intervenção do usuário.

3.2 Após a análise de todas as condições de parada, aplicar a seguinte função para que a solução final seja encontrada:

. **IL = Mínimo(Solução para I1, ..., Solução para In,**
Solução para O1, ..., Solução para Om,
Máximo(Sol. para Di1, ..., Sol. para Dii),
...
Máximo(Sol. para Dj1, ..., Sol. para Djl))

3.3 Substituir IL nas soluções encontradas na Parte 2 e que são dependentes do número de ciclos.

Em resumo, este algoritmo tenta representar o efeito do "loop", derivando expressões simbólicas para cada variável definida em seu corpo. Porém, cabe indagar o que acontecerá quando uma solução não for encontrada. Neste caso, o usuário deverá desenvolver e informar uma função que capture o valor da variável sob análise e síntese. O Apêndice D apresenta uma série de funções que podem ser utilizadas com este fim, utilizando-se, até mesmo, funções recursivas que traduzem o comportamento do "loop". Todavia, o usuário é livre para escolher a melhor forma de indicar esta funcionalidade, tendo como única restrição a concordância da solução obtida com a sintaxe definida na Seção 4.2.4.

4.5 Subsistema "Expansor de Abstração Funcional"

O último subsistema da FAF tem por objetivo facilitar a interpretação do resultado obtido pelo subsistema "Gerador de Abstração Funcional". No estágio atual, ele realiza a expansão de uma abstração funcional, que esteja na forma intermediária, para uma expressão baseada nos construtores matemáticos e funcionais. Apesar da sua importância, a funcionalidade deste subsistema implementado é muito simples e necessita de simplificadores algébricos, os quais deverão ser aplicados às expressões finais obtidas.

4.6 Resumo dos Processos Executados pela FAF

Esta seção descreve os processos existentes nos vários subsistemas que compõem a FAF, visando sumarizar o que a FAF executa quando um programa alvo está sendo analisado e sintetizado. Neste sentido, os processos executados pela FAF podem ser resumidos em sete passos:

1. Geração de uma forma intermediária do programa, representada por uma árvore "And-Or", na qual os nós ("Call", "Nill", "Loop", "Or" e "And") traduzem os vários construtores utilizados na linguagem. Observe que cada nó "nill" e cada condição do programa são representados por expressões binárias com um operador e dois

operandos. Ademais, uma tabela de símbolos faz parte desta forma intermediária.

2. Análise de todas as variáveis do programa, objetivando gerar um conjunto com suas Variáveis de Estado (Capítulo 3). Atualmente esta análise é feita manualmente.

3. Segmentação do programa alvo através da utilização do "Slicer" em termos de suas variáveis relevantes para abstração (Variáveis de Estado), gerando uma subárvore "And-Or" correspondente aos comandos que, de alguma maneira, afetam o comportamento do conjunto de variáveis que se quer analisar.

4. Obtenção da decomposição do programa em primos a partir da subárvore gerada em 3.

5. Síntese de cada primo de acordo com o seu tipo, associando uma função a cada um deles. A abstração dos programas primos é obtida pela aplicação dos algoritmos descritos neste capítulo.

6. Composição funcional das abstrações parciais (primos), objetivando obter a funcionalidade do programa alvo.

7. Expansão da abstração final com o intuito de facilitar a sua interpretação pelo usuário da FAF.

Todos estes passos foram, de alguma forma, implementados na Ferramenta de Abstração Funcional discutida neste capítulo(7). Entretanto, faz-se relevante ressaltar que esta ferramenta é apenas um protótipo, o qual precisa ser aperfeiçoado e expandido para incorporar funções que permitam a análise e abstração de programas reais.

---

(7) O código fonte da FAF encontra-se descrito em [EVAN-92]. Relatório produzido pelo autor para documentar a implementação da ferramenta.

4.7 Limitações do Protótipo

A ferramenta de Abstração Funcional proposta neste trabalho de pesquisa possui algumas limitações, quais sejam: não trabalha com "array", com ponteiros, com chamadas às funções recursivas e iterações que controlam leituras de arquivos.

4.8 Considerações Finais

Este capítulo apresentou a Ferramenta de Abstração Funcional (FAF) em seus aspectos mais importantes e descreveu os algoritmos e as estruturas utilizadas em sua implementação.

Esta ferramenta foi descrita com o objetivo de organizar e automatizar o processo de abstração de um código fonte, sendo a sua base formada por quatro conceitos: a preparação do código fonte, a segmentação de programas, a composição funcional e a análise das expressões de recorrência em "loops".

A preparação do código fonte foi definida como a atividade de obtenção ou de geração de informações relevantes do programa para facilitar a tarefa de abstração, como, por exemplo, a obtenção de tabela de símbolos, a determinação do escopo das variáveis, a determinação das variáveis definidas e referenciadas em cada comando, a geração de uma forma intermediária para representar o programa e os seus comandos, a padronização dos comandos de decisão, etc.

A segmentação de programas foi utilizada com o objetivo de identificar as partes constituintes do programa, assumindo-se que cada parte é mais fácil de ser entendida do que o todo. Neste contexto, a técnica de "Slicer" e a decomposição em programa primos foram utilizadas para a determinação destes segmentos e da hierarquia de primos. Esta segmentação não só permite uma maior flexibilidade em relação ao objeto a ser analisado, como organiza e sistematiza o processo de abstração com o intuito de se conseguir resultados mais concisos.

Finalmente, a composição funcional (seqüência), a geração de casos (decisão) e a análise de recorrência de "loops" foram utilizadas para permitir a análise semi-automática de cada programa primo, de modo que a funcionalidade do programa possa ser obtida passo a passo.

Pode-se afirmar que a ferramenta de abstração proposta neste trabalho permitiu a avaliação e a consolidação do modelo proposto no Capítulo 3, apesar de necessitar de vários tratamentos para possibilitar a sua utilização em programas reais, quais sejam: tratamento de ponteiros, "arrays" e aumento da automação em relação aos comandos de decisão e iteração. Cabe ressaltar que não existem ferramentas no mercado direcionadas à obtenção da abstração funcional de programas, inviabilizando comparações.

Capítulo 5

Verificação Prática da Ferramenta de Abstração Funcional

Este capítulo evidencia que os resultados gerados pela Ferramenta de Abstração Funcional (FAF) estão corretos, ou seja, produzem os mesmos valores que o programa alvo quando aplicados a um conjunto de dados de entrada. Ademais, este capítulo pretende mostrar a flexibilidade desta ferramenta em duas situações diferentes.

5.1 Verificação da FAF

Esta seção é constituída de dois exemplos que objetivam a verificação da FAF através da sua aplicação prática. O primeiro exemplo descreve como a FAF é ativada e, principalmente, evidencia que os resultados da abstração funcional de um programa alvo estão corretos. O segundo exemplo analisa um programa que só pode ser abstraído parcialmente.

5.1.1 Exemplo 1: Ativação da FAF e Validação dos Resultados

O programa mostrado na Figura 5.1 tem por objetivo calcular o tempo e a distância percorrida por um navio até que sua velocidade seja reduzida a zero durante a aproximação a um cais.

```
    extern double distance, time;
    extern int    error;
    void Dck(station, starship, thrust, velocity, deltat)
    double station, starship, thrust, velocity, deltat;
    {
 1  double gconst, gravity, constacc, curvel, nextvel;
 2
 3  if (station <= 0.0 || starship <= 0 || thrust <= 0 ||
        velocity <= 0 || deltat <= 0.0 || time <= 0.0 ||
        distance <= 0.0) {
 4      error = 1;
 5  }
 6  else {
 7       gconst = 6.67 * 10.e-11;
 8       gravity  = gconst * station * starship /(distance*distance);
 9       constacc = gravity - thrust / starship;
10       curvel = velocity;
11       nextvel = curvel + constacc * deltat;
12
13       do {
14            distance = distance - curvel * deltat;
15            curvel   = nextvel;
16            time     = time + deltat;
17            nextvel = curvel + constacc * deltat;
18       } while(nextvel > 0.0);
19       error = 0;
   }
```

Figura 5.1: Programa "DOCKING" [MUCH-81]

Neste exemplo, os parâmetros formais e as variáveis globais possuem o seguinte significado:

Parâmetros Formais:

- station : massa da base de destino do navio (kg)
- starship: massa do navio (kg)
- thrust : força de empuxo do navio (nt)
- velocity: velocidade inicial do navio (m/s)
- deltat : valor para incrementação do tempo (s)

Variáveis Globais:

- time : tempo inicial e final (s)
- distance: distância inicial e final entre o navio e a sua base (m)

O programa ilustrado na Figura 5.1 será analisado e sintetizado para as suas Variáveis de Estado "distance" e "time" (variáveis globais). Observe que este programa já é estruturado, não necessitando, desta forma, passar por uma ferramenta de estruturação.

Geração da "Representação Intermediária" do Programa "Docking"

Em primeiro lugar, deve-se obter a forma intermediária para o programa, executando-se o seguinte subsistema da ferramenta com o seu respectivo parâmetro:

Andtree -iDck(1)

(1) A opção -i é utilizada para indicar o nome do arquivo a ser analisado (neste caso, o programa "DCK.C").

Quatro arquivos "ASCII" serão gerados para o programa dado[2]. Os nomes destes arquivos serão formados por "Dck", seguidos por uma extensão que indique o tipo do arquivo. Para o exemplo dado, tem-se:

Dck.atf : Arquivo que armazena todas as informações necessárias para se criar uma árvore "And-Or". Esta árvore representa o fluxo de controle e de dados do programa e, além disto, possui ponteiros para a forma intermediária dos comandos do programa a ser analisado e sintetizado.

Dck.var : Arquivo que representa a tabela de símbolos do programa. Cada linha deste arquivo possuirá o nome da variável, o seu escopo e o seu tipo.

Dck.eqn : Arquivo que armazena a forma intermediária de todos os comandos de atribuição do programa, bem como os comandos condicionais nele encontrados.

Dck.con : Arquivo que armazena todas as constantes encontradas no programa.

Análise e Síntese do Programa "Docking"

O protótipo implementado não verifica automaticamente quais são as Variáveis de Estado. Todavia, o usuário deverá fornecer uma lista de variáveis que seja de seu interesse para a análise. Neste caso, a sintaxe necessária para ativar a ferramenta de abstração em relação ao conjunto de Variáveis de Estado é descrita abaixo:

Abstrai -iDck -n1,19 -gtime,distance[3],

onde a opção "-i" indica o nome do arquivo a ser analisado, ou seja, o arquivo que armazena a árvore "And-Or", a opção "-n" indica as linhas que deverão ser abstraídas e a opção "-g" indica a lista de variáveis globais

---

[2] O apêndice E contém a descrição e o conteúdo destes arquivos para o exemplo em estudo.

[3] Esta ferramenta ativa o subsistema de "Análise de Decomposição" e o subsistema "Gerador de Abstração Funcional".

a serem analisadas. Uma opção "-l" pode ser também utilizada com o objetivo de indicar a lista de variáveis locais de interesse.

A ferramenta em questão ativará o subsistema de segmentação para extrair automaticamente os nós da árvore "And-Or" que afetam o critério dado como entrada (número das linhas e variáveis). Após a extração com o "Slicer", esta subárvore gerada deverá ser decomposta em termos de seus programas primos. A extração com "Slicer" e a decomposição em primos é apresentada na Figura 5.2.

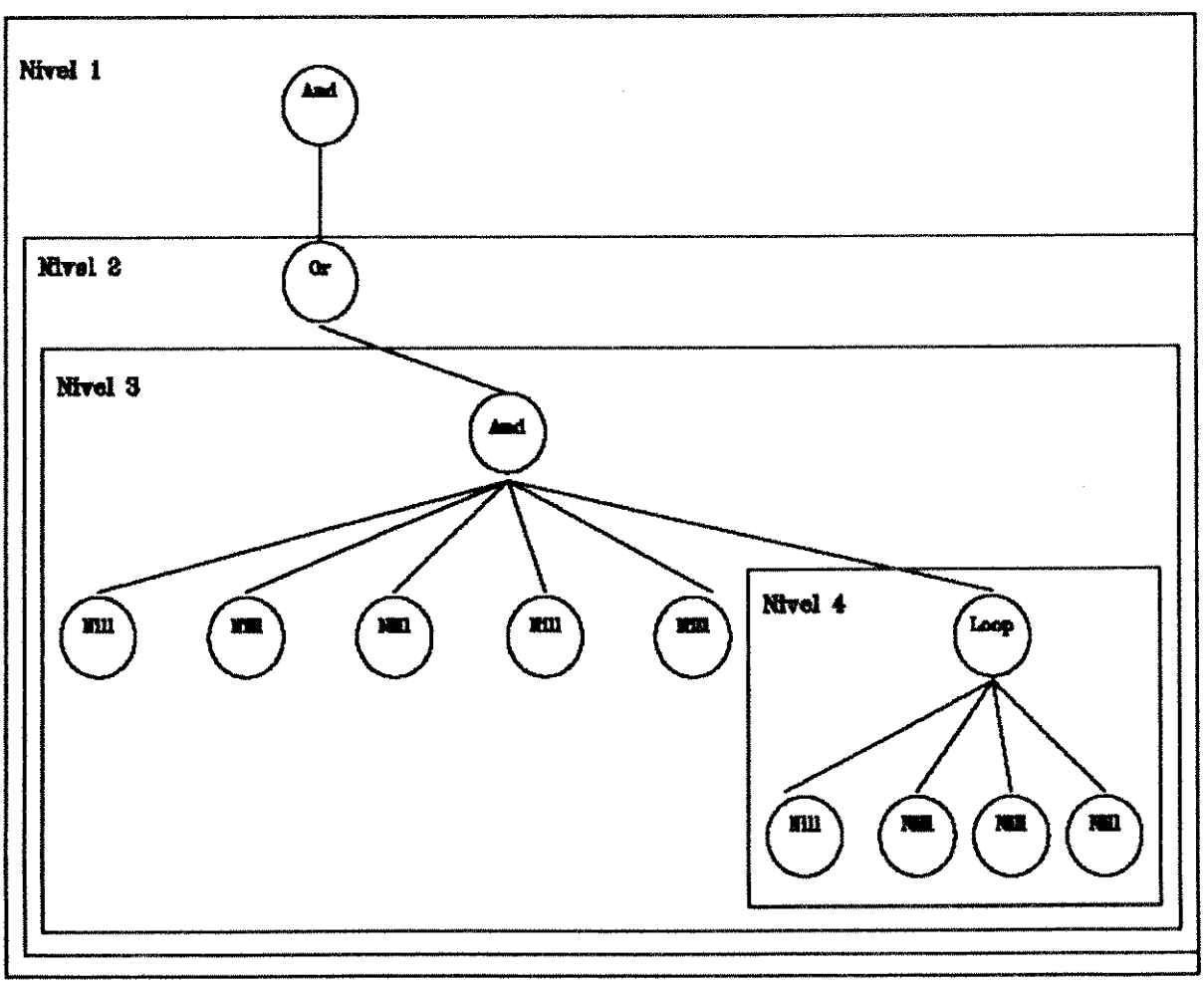


Figura 5.2: Decomposição do Programa "Docking" para a Variável "time" e "distance".

A abstração funcional do programa será efetuada de acordo com a hierarquia de primos apresentada na Figura 5.2, o que produz como resultado final as expressões listadas na Figura 5.3[4].

CASO 1: station $\leq$ 0.0 OR starship $\leq$ 0.0 OR thrust $\leq$ 0.0 OR
velocity $\leq$ 0.0 OR deltat $\leq$ 0.0 OR distance $\leq$ 0.0

time := time

distance := distance

CASO 2: station $>$ 0.0 AND starship $>$ 0.0 AND thrust $>$ 0.0 AND
velocity $>$ 0.0 AND deltat $>$ 0.0 AND distance $>$ 0.0

time = time + IL * deltat

$$\text{distance} = \text{distance} - \sum_{i=1}^{IL} \left[(\text{velocity} * \text{deltat}) + \sum_{j=2}^{i} \left((6.67 * 10^{-11} * \text{station} * \text{starship} / \text{distance}^2 - \text{thrust} / \text{starship}) * \text{delta}^2 \right) \right]$$

ONDE:

IL = 1 + Floor[Abs (N)]

$$N = \frac{\text{deltat} * \left[\frac{6.67 * 10^{-11} * \text{starship} * \text{station}}{\text{distance}^2} - \frac{\text{Thrust}}{\text{starship}} \right] + \text{velocity}}{\text{deltat} * \left[\frac{6.67 * 10^{-11} * \text{starship} * \text{station}}{\text{distance}^2} - \frac{\text{Thrust}}{\text{starship}} \right]}$$

Figura 5.3: Abstração Funcional do Programa "Docking" em Relação às Variáveis "time" e "distance".

---

[4] O apêndice F retrata detalhadamente a análise de abstração efetuada sobre cada programa primo deste exemplo.

Esta abstração funcional indica que o programa poderá produzir dois resultados diferentes para cada uma das variáveis "time" e "distance". No primeiro caso, onde existe pelo menos uma variável ("station", "starship", "thrust", "velocity", "deltat", "time" ou "distance") cujo valor seja menor ou igual a zero, os resultados das variáveis "time" e "distance" serão os seus valores de entrada.

No segundo caso, onde todas as variáveis de entrada são maiores do que zero, as variáveis "time" e "distance" assumirão os valores listados no "CASO 2" da Figura 5.3. Observe que todas as expressões geradas são dependentes do número de ciclos do "loop" (IL) e foram simplificadas para facilitar o seu entendimento.

Validação do Resultado

A estratégia escolhida para validar o resultado obtido pela FAF para o programa "Docking" consiste na demonstração de que o resultado obtido pelo programa alvo, quando executado com um determinado conjunto de dados de entrada, não difere do resultado obtido pela aplicação dos mesmos dados de entrada à abstração gerada pela FAF. Neste sentido, a Tabela 5.1 contém 10 testes executados com este fim.

Nesta tabela, cada linha corresponde a um novo teste efetuado. As colunas de 1 até 5 correspondem aos valores dos parâmetros de entrada do programa, as colunas 6 e 7 correspondem aos valores das variáveis "time" e "distance" gerados pelo programa "Docking" para os dados de entrada, a coluna 8 indica o número de ciclos executados pelo programa em seu "loop" interno, as colunas 9 e 10 representam os valores das variáveis "time" e "distance", obtidos pela aplicação dos valores de entrada às fórmulas da Figura 5.3, finalmente, a coluna 11 representa o número de ciclos calculados a partir da fórmula de **IL**, apresentada pela mesma figura.

station	starship	thrust	velocity inicial	deltat	Ciclos Loop Docking	Distance Final Docking	Time Final Docking	Ciclos Calculado Abstração	Distance Final Abstração	Time Final Abstração
5000	500	500	30	0.10	300	48.5	31	300	48.5	31
50000	500	500	30	0.10	300	8.5	31	300	8.5	31
50000	500	500	40	0.10	400	8	41	400	8	41
500000	5000	5000	50	0.01	5000	49.7488	51	5000	49.7488	51
5000000	500000	500000	30	0.20	150	245.4788	31	150	245.4788	31
5.0E+08	50000000	50000000	50	0.20	338	6304.563	68.6	338	6304.563	68.6
20000000	200000	700000	35	0.10	100	522.9805	11	100	522.9805	11
2.0E+08	700000	700000	35	0.10	409	81.1217	41.9	409	81.1217	41.9
20000000	200000	100000	50	0.10	1000	496.0193	101	1000	496.0193	101
20000000	200000	150000	30	0.01	4001	2399.612	41	4001	2399.612	41

Tabela 5.1: Comparação dos Resultados Obtidos pela Aplicação dos Parâmetros de Entrada ao Programa "Docking" e à Abstração Funcional Correspondente.

Pode-se observar que os resultados não diferem em nenhuma das situações testadas, o que é uma ótima indicação de que as expressões geradas pela FAF estão corretas para o programa "Docking".

5.1.2 Exemplo 2: Abstração Parcial de um Programa

A FAF não está preparada para reconhecer e abstrair corretamente iterações que são dependentes de leituras ou gravações de arquivos. Apesar desta limitação atual, a FAF pode auxiliar no processo de obtenção da abstração funcional de programas que utilizam estes construtores.

Considere o exemplo da Figura 5.4, que possui duas iterações aninhadas. A iteração mais externa não pode ser sintetizada, pois controla a leitura e a gravação de um arquivo, sendo que todo o cálculo deste programa é feito no bloco interno ao "loop" externo, o qual é passível de abstração.

```
#include <stdio.h>

struct arq {
    int    acct_num;
    double balance;
    double loan_out;
    double loan_max;
     char   default;
};

type arq adjust_rec;
type arq acc_rec;
int  csf_at_end;

main() {

 1      File *fd, *fd1;
 2      char default;
 3      fd = fopen("account_file","r");
 4      fd1= fopen("adjustment_rec","w");
 5
 6      csf_at_end = 'f';
 7
 8      while(csf_at_end == 'f') {
 9
10      fscanf(fd, "%d %f %f %f\n",&acc_rec.acct_num, &acc_rec.balance,
                                   &acc_rec.loan_out,
&acc_rec.loan_max);
11
12      adjust_rec.acct_num = acc_rec.acct_num;
13      adjust_rec.balance  = acc_rec.balance;
14      adjust_rec.loan_out = acc_rec.loan_out;
15      adjust_rec.loan_max = acc_rec.loan_max;
16
17       if (adjust_rec.balance < 0.00)
18            default = 0;
19       else
20            default = 1;
21
22      while(adjust_rec.balance < 0.00 &&
23            adjust_rec.loan_out + 100.00 < adjust_rec.loan_max) {
24                 adjust_rec.loan_out += 100.0;
25                 adjust_rec.balance += 100.0;
26      }
27      fprintf(fd1, "%f %f\n", adjust_rec.loan_out,
                                adjust_rec.balance);
28      }
29      fclose(fd);
30      fclose(fd1);
   }
```

Figura 5.4: Programa com Dois "Loops" Aninhados [HAUS-90]

A abstração do bloco de programa relacionado ao "loop" mais externo é obtida utilizando-se a seguinte sintaxe:

```
Abstrai -iarquivo -n12,26 -gadjust_rec.loan_out,
                          adjust_rec.balance
```

O resultado da abstração é ilustrado na Figura 5.5. Observe que esta abstração possui uma parte informal (texto dentro do quadro) criada manualmente pelo autor desta pesquisa, que corresponde ao "loop" externo.

Para cada registro do arquivo "Account_File" é gravado um registro no arquivo "Adjustmene-File", cujos campos correspondem aos valores das variáveis "adjust_rec.loan_out" e "adjust_rec.balance". Os valores assumidos por estas variáveis são descritos em dois casos:

CASO 1: (ajust_rec.balance >= 0.00 OR
adjust_rec.loan_out + 100 >= adjust-rec.loan-max)

adjust_rec.loan_out := adjust_rec.loan_out

adjsut_rec.balance := adjust_rec.balance

CASO 2: (ajust_rec.balance < 0.00 AND
adjust_rec.loan_out + 100 < adjust_rec.loan_max)

adjust-rec.balance := acc-rec.balance + (100 * IL)

adjust-rec.loan-out := acc-rec.loan + (100 * IL)

Onde:

IL := minimo[ceiling((0.0 - acc_rec.balance) / 100.0),
1+floor((acc_rec.loan_max - 100.0 -
acc_rec.loan_out)/100.0)]

Figura 5.5: Abstração Funcional do Exemplo da Figura 5.4.

Este exemplo mostrou que a FAF é útil também para analisar e abstrair programas que ainda não podem ser totalmente suportados pela ferramenta. O Apêndice G fornece mais um exemplo para demonstrar as principais interfaces da FAF.

5.2 Considerações Finais

Este capítulo apresentou dois exemplos que foram utilizados com o objetivo de avaliar a FAF através da sua aplicação prática. O primeiro foi apresentado com o intuito de mostrar os passos necessários para utilização da FAF e, em adição, evidenciar que o resultado da abstração estava correto para um determinado número de testes. O segundo destacou a flexibilidade da ferramenta na análise e abstração de uma parte do programa alvo. Esta flexibilidade é especialmente importante quando o programa alvo não puder ser totalmente abstraído.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste capítulo são apresentadas conclusões sobre este trabalho de pesquisa e recomendações sobre trabalhos futuros. Neste sentido, na Seção 6.1 são discutidas as conclusões e aplicações e na Seção 6.2 são propostos vários tópicos de trabalhos de pesquisa futuros.

6.1 Conclusões

Este trabalho de pesquisa estudou, desenvolveu e implementou um modelo para abstração funcional, cujo objetivo é a determinação precisa do efeito de um programa, ou parte dele, sobre as suas variáveis nas diversas situações encontradas no transcorrer do seu fluxo de controle. Vale apenas observar, que estas situações são representadas por regras ou casos de execução, formados por um domínio (condição de execução do caso) e por uma série de expressões simbólicas, que determinam o efeito do programa sobre os seus dados.

Para atingir tal objetivo foi escolhida uma estratégia "Bottom-Up", com o intuito de derivar-se a funcionalidade do programa, baseando-se nas informações contidas no código fonte. Ademais, o modelo apresentado para determinação da abstração funcional de programas foi fundamentado na aplicação dos conceitos da programação estruturada, na teoria funcional e nas técnicas de execução simbólica, de simplificações algébricas e de "Slicer" de programas.

Os conceitos da programação estruturada foram estudados para sistematizar o processo de obtenção da funcionalidade de um programa através da estratégia "Bottom-Up", de acordo com a proposta de Hausler [HAUS-90]. Neste contexto, utilizou-se a segmentação de programas primos, cujas propriedades permitem o seu entendimento isolado e a sua composição funcional com outros primos, viabilizando a derivação gradativa da função executada pelo programa.

As técnicas de execução simbólica foram estudadas para tornar o modelo de Hausler mais consistente e automatizável. Nesta direção, buscou-se na técnica de execução simbólica global [CHEA-79] fundamentos práticos de padronização dos comandos de decisão, das técnicas para simplificação algébrica e da análise das relações de recorrência do construtor para iteração, esta última, associada à segmentação da iteração em função de seus componentes básicos (computações independentes, filtros e "loop" básico) [WATE-79].

Uma das contribuições deste trabalho de pesquisa a nível de mestrado foi a utilização da técnica de "Slicer" para decomposição de programas com o intuito de permitir a definição e a incorporação de um novo conceito intitulado "Variáveis de Estado" ao modelo de abstração funcional apresentado neste trabalho. Neste sentido, foi possível demonstrar que um programa pode ser analisado e sintetizado em função das variáveis que efetivamente capturam o seu comportamento, diminuindo a sua complexidade através de sua segmentação em pedaços menores e mais fáceis de serem entendidos.

Entretanto, a principal contribuição deste trabalho de pesquisa foi o estabelecimento de um modelo de abstração funcional baseado nos trabalhos de Hausler et al [HAUS-90], Cheatham et al [CHEA-79] e Waters [WATE-79]. O modelo de Hausler proporcionou o fundamento básico do processo de abstração e os trabalhos de Cheatham e Waters preencheram lacunas deixadas por Hausler (por exemplo, padronização dos comandos e algoritmos para simplificação e determinação da factibilidade dos comandos de decisão).

O modelo de abstração funcional aqui apresentado foi validado e consolidado através da implementação de uma ferramenta que utiliza o código fonte de um programa como única fonte de informação. Ele propicia a geração de uma forma intermediária do programa para facilitar a atividade de abstração, a segmentação do programa em função de suas Variáveis de Estado, a decomposição do segmento encontrado em programas primos e, a análise e a síntese de cada um dos primos.

O processo de análise de abstração dos primos que representam seqüências utilizou as técnicas de "Trace-Table" e "Trace-Table" condicional. Por sua vez, o processo de análise dos primos que representam decisões envolveu a geração de casos para a parte "then" e para a parte "else" do primo em questão. Finalmente, o processo de análise dos primos que representam iterações abrangeu a sua segmentação em termos de seus elementos computacionais básicos e a resolução das relações de recorrência encontradas.

Vale ressaltar que a Ferramenta de Abstração Funcional (FAF) implementada possui meios para determinar, em casos mais triviais, se uma determinada condição de execução gerada no processo de análise é ou não factível e, ainda, permitir a abstração em mais alto nível de um primo que represente uma decisão (neste caso, com o auxílio do usuário). Estas duas características foram incorporadas com o objetivo de reduzir o número de casos gerados pela ferramenta.

A Ferramenta de Abstração Funcional (FAF) desenvolvida nesta pesquisa pode ser potencialmente utilizada nas atividades de manutenção, teste, documentação, verificação, especialização, depuração e desenvolvimento.

- Manutenção: a Ferramenta de Abstração Funcional desenvolvida pode fornecer resultados que facilitam o entendimento do programa. Este entendimento permite que se determine "onde" e "como" as mudanças requeridas serão efetuadas [HAUS-90]. Quando as mudanças estiverem concluídas, o programador poderá submeter novamente o programa à ferramenta com o intuito de derivar as novas funções computadas pelo programa, assegurando, assim, que as modificações estão corretas e não acarretam efeitos colaterais.

- Teste: as condições envolvidas no resultado obtido pela Ferramenta de Abstração Funcional podem ser utilizadas para auxiliar a determinação de dados de entrada para casos de teste de um programa.

- Documentação: a Ferramenta de Abstração Funcional poderá ser utilizada para documentar um programa já totalmente implementado. Esta documentação será útil para as atividades de manutenção e evolução [HAUS-90].

- Verificação: dado o resultado obtido pela Ferramenta de Abstração Funcional e a especificação externa acerca do comportamento esperado do programa, é possível verificar se o mesmo está ou não de acordo com a sua especificação [HAUS-90].

- Especialização: a Ferramenta de Abstração Funcional poderá ser utilizada na especialização de um programa genérico, permitindo que este execute uma determinada funcionalidade requerida, isto é, um conjunto de casos de execução, restringindo-se o domínio de atuação do programa [COEN-91]. Esta especialização pode ser vista como uma partição funcional do programa, pois o objetivo é retirar os comandos do programa que afetam um conjunto de casos de execução de interesse.

- Depuração: o resultado obtido pela Ferramenta de Abstração Funcional poderá ser utilizado para determinar a causa de um erro encontrado no programa [CLAR-85].

- Desenvolvimento: a Ferramenta de Abstração Funcional poderá ser utilizada para verificar se o programa em desenvolvimento está de acordo com a funcionalidade esperada [CLAR-85, HAUS-90].

Enfim, pode-se afirmar que este trabalho de pesquisa a nível de mestrado contribuiu para a definição de um modelo de abstração funcional promissor, fornecendo uma fundamentação teórica necessária à análise automática de programas complexos. Entretanto, os resultados deste trabalho representam apenas um embrião de outras atividades de pesquisa que poderão ainda ser desenvolvidas.

6.2 Trabalhos Futuros

A Ferramenta de Abstração Funcional baseada no modelo apresentado pode ser útil para algumas fases do ciclo de vida clássico de um software (por exemplo, manutenção e desenvolvimento), o que garante a sua relevância e, portanto, o interesse na expansão do trabalho aqui realizado.

Neste contexto, cabe a sugestão dos seguintes trabalhos de pesquisa futuros:

- Simplificação do resultado final obtido pela Ferramenta de Abstração Funcional com o objetivo de tornar esta abstração mais concisa e interpretável. Tais simplificações são comparáveis àquelas utilizadas em sistemas de manipulação algébrica [BUCH-82];

- Aumento da capacidade de reconhecimento e eliminação de casos impossíveis pela Ferramenta de Abstração, já que um dos principais problemas encontrados na abstração de um programa é a determinação da consistência das condições envolvidas nos casos de execução. Vale destacar a importância da determinação desta consistência o mais cedo possível, evitando-se o dispêndio de recursos computacionais na avaliação de casos não factíveis;

- Determinação da forma de representação e de manipulação de um "array", já que a determinação dos valores relacionados com os subscritos de um "array" é um problema não abordado no modelo de abstração proposto;

- Expansão da técnica para análise do construtor de iteração com o objetivo de cobrir um número maior de iterações utilizadas em programas reais. Um caminho recomendado, em adição à técnica de análise de recorrência, é o reconhecimento de padrões de comportamento das iterações para a determinação de sua funcionalidade [HAUS-90, WATE-79];

- Inclusão de heurísticas que permitam transformar um primo com "n" casos de execução em uma abstração que produza um simples valor; e, finalmente,

- Tratamento de ponteiros e determinação dos comandos que afetam uma expressão simbólica específica gerada, que podem e devem ser incorporados à Ferramenta de Abstração Funcional desenvolvida neste trabalho de pesquisa.

Referências:

[BART-77] Bartussek W., Parnas D. L., *Using Traces to Write Abstract Specifications for Software Modules*, University of North Carolina, Chapel Hill, NC, Report TR 77-012, December 1977, pp. 1-20.

[BASI-75] Basili V. R., Noonan R. E., *A Comparison of the Axiomatic and Functional Models of Structured Programming*, Department of Computer Science, University of Maryland, Technical Report, 1981, pp. 1-42.

[BASI-82] Basili R., Mills H. D., *Understanding and Documenting Programs*, IEEE Trans. on Software Engineering, May 1982, pp. 270-283.

[BIGG-89] Biggerstaff T. J., *Design Recovery for Maintenance and Reuse*, Computer, July, 1989, pp. 36-49.

[BISH-90] Bishop J. M., *The Effect of Data Abstraction on Loop Programming Techniques*, IEEE Trans. on Software Engineering, vol. 16, no. 4, April 1990, pp. 389-402.

[BUCH-82] Buchberger B., Collins G. E., Loos R., *Computer Algebra: Symbolic and Algebraic Computation*, Springer-Verlag, New York, 1982.

[BYRN-91] Byrne E. J., *Software Reverse Engineering: A Case Study*, Software Practice and Experience, vol. 21 (12), December 1991, pp. 1349-1364.

[CHEA-79] Cheatham T. E., Holloway G. H., Townley J. A., *Symbolic Evaluation and the Analysis of Programs*, IEEE Trans. on Software Engineering, vol. SE-5, no. 4, July 1979, pp. 402-417.

[CHIK-90] Chikofsky E. J., Cross II J. H., *Reverse Engineering and Design Recovery: A Taxonomy,* IEEE Software, January, 1990, pp. 13-17.

[CLAR-76] Clarke L. A., *A System to Generate Test Data and Symbolically Execute Programs,* IEEE Trans. on Software Engineering, vol. 2, September 1976, pp. 215-222.

[CLAR-85] Clarke L. A., Richardson D. J., *Applications of Symbolic Evaluation,* The Journal of Systems and Software, no. 5, 1985, pp. 15-35.

[COEN-91] Coen P., Paoli A., Ghezzi C., Mandrioli D., *Software Specialization Via Symbolic Execution",* IEEE Trans. on Software Engineering, vol. 17, no. 9, September 1991, pp. 884-899.

[CORB-89] Corbi T. A., *Program Understanding: Challenge for the 1990's,* IBM Systems Journal, vol. 28, no. 2, 1989, pp. 294-306.

[CURT-85] Curtis B., *Human Factors in Software Development,* IEEE Computer Society Press, Washington, DC, 1985.

[EVAN-90b] Evangelista S. R. M., Moura M. F., Nascimento M., Ternes S., *Especificação Formal da "And-Or-Tree" por Traço,* Campinas, SP, EMBRAPA/NTIA, Relatório Interno, 1990.

[EVAN-92] Evangelista S. R. M. *Implementação da Ferramenta de Abstração Funcional,* Campinas, SP, EMBRAPA/NTIA, Relatório Interno, 1992.

[FAIR-75] Fairley R. E., *An Experimental Program Testing Facility,* IEEE Trans. on Software Engineering, vol. SE 1, December 1975, pp. 350-357.

[HAUS-90] Hausler P. A., Pleszkoch M. G., Linger R. C., Hevner A. R., *Using Function Abstraction to Understand Program Behavior*, IEEE Software, January, 1990, pp. 55-63.

[HUAN-75] Huang J. C., *An Approach to Program Testing System*, ACM Comput. Surv., vol 7, no. 3, September 1975, pp. 113-128.

[IEEE-83] *IEEE Standart Glossary of Software Engineering Terminology*, New York, 1983, (ANSI/IEEE std. 729/83)

[KERN-78] Kernighan B. W., Ritchie D. M., *The C Programming Language*, Prentice-Hall, New Jersey, 1978.

[KEUF-90] Keuffel Warren, *Making Sense of It All: Groupware for Re-engineering*, Computer, June, 1990, pp. 93-101.

[KING-76] King J. C., *Symbolic Execution and Program Testing*, Communication of the ACM, vol. 19, no. 7, July 1976, pp. 385-394.

[JOHN-84] Johnson S. C., *A Tour Through "C" Portable Compiler*, ULTRIX-32 Supplementary Documentary, Vol 2, pp. 237-261, 1984.

[LEHN-80] Lehnman M. M., *Programs, Life Cycles and Laws of Software Evolution*, Proc. of IEEE, Vol. 68, no 9, September 1980.

[LING-79] Linger R. C., Mills H. D., Witt B. I., *Structured Programming Theory and Practice*, Addison-Wesley Publ. Co., Cambridge, Mass., 1979.

[MILL-75] Mills H. D., *The New Math of Computer Programming*, Communications of the ACM, vol. 18, no. 1, January-1975, pp. 43-48.

[MOUR-92] Moura M. F., *Tese de Mestrado*, DCA/FEE/UNICAMP - Campinas, SP, Brasil, 1992 (em preparação).

[MUCH-81] Muchnick S. S., Jones N. D., *Program Flow Analysis Theory and Applications*, Prentice-Hall, New Jersey, 1981.

[PRAT-78] Pratt T. W., *Control Computations and the Design of Loop Control Structures*, IEEE Trans. on Software Engineering, vol. SE-4, no. 2, March 1978, pp. 81-89.

[REZE-92] Rezende M. C., *Descrição do Algoritmo de Slicer*, Campinas, SP, EMBRAPA/NTIA, Relatório Interno, 1992.

[RICH-90] Rich C., Wills L. M., *Recognizing a Program's Design: A Graph-Parsing Approach*, IEEE Software, January 1990, pp. 82-89.

[SCAN-90] Scandura J. M., *Cognitive Approach to Systems Engineering and Re-engineering: Integrating New Designs with Old Systems*, Software Maintenance: Research and Practice, Vol. 2, pp. 145-156, 1990.

[SCHI-91] Schildt H., *C Completo e Total*, McGraw-Hill, Rio de Janeiro, RJ, 1991.

[SHNE-79] Shneiderman B., Mayer R., *Syntactic/Semantic Interactions in Programmer Behavior: A model and Experimental Results*, International Journal of Computer and Information Sciences, vol. 8, no. 2, 1989, pp. 219-238.

[SOBR-84] Sobrinho F. G., *Structural Complexity: A Basis For Systematic Software Evolution*, Ph.D. Thesis, Department of Computer Science, University of Maryland, 1984.

[STEP-92] Stephen C. J., *Yacc - Yet Another Compiler Compiler*, ULTRIX-32(TM) Supplementary Documents Volume II, Bell Laboratories, 1992.

[ZELK-90] Zelkowitz M. V., *Model of Program Verification,* IEEE Computer, September 1990, pp. 30-39.

[WATE-79] Waters R. C., *A Method for Analyzing Loop Programs,* IEEE Trans. on Software Engineering, vol 5, no 3, May 1979, pp. 237-247.

[WEIS-82] Weiser M., *Programmers Use Slicing when Debugging,* Communications of the ACM, vol. 25, no. 7, July 1982,. pp. 446-452.

[WEIS-84] Weiser M., *Program Slicing,* IEEE Trans. on Software Engineering, vol. SE-10, July 1984, pp. 352-357.

[WIRT-78] Wirth N., *Programação Sistemática,* Editora Campus LTDA, Rio de Janeiro, RJ, 1978.

[YEH-90] Yeh R. T., *An Alternative Paradigm for Software Evolution,* In: Ng P. A. & Yeh R. T. (editors), Modern Software Engineering; foundations and current perspectives, New York, Van Nostrand Reinold, pp. 7-22, 1990.

Apêndice A

Especificação em YACC da Linguagem "C"

Este apêndice apresenta a especificação em YACC da linguagem "C" utilizada para reconhecer um programa alvo escrito nesta linguagem. Esta especificação é baseada em [JONH-84] e é suficiente para mostrar quais são os construtores suportados, quais são as expressões válidas e qual é a prioridade de avaliação das expressões, etc.

```
/* ------------------------------------------------------------------------
```

1) Símbolos Terminais (Token):

Operadores Simples:

		Associatividade:
PLUS	+	Esquerda
MINUS	-	Esquerda
MUL	*	Esquerda
AND	&	Esquerda
OR	!	Esquerda
ER	^	Esquerda
QUEST	?	Direita
COLON	:	Direita
ANDAND	&&	Esquerda
OROR	!!	Esquerda

Operadores que são abreviações:

ASOP	+=, -=, *=, /=, &= !=, ^=, %=, >>=, <<=	Direita
RELOP	<=, >=, >, <	Esquerda
EQUOP	==, !=	Esquerda
DIVOP	/, %	Esquerda
SHIFTOP	>>, <<	Esquerda
INCOP	++, --	Direita
UNOP	! (Not) , ~ (Compl.)	Direita
STROP	., ->	Esquerda

Separadores

LP	(	Esquerda
RP	)	-
LC	{	-
RC	}	-
LB	[	Esquerda
RB	]	-
CM	,	Esquerda
SM	;	-
ASSIFN	=	Direita

```
 * ------------------------------------------------------------------------ */

%start ext_def_list

%type <intval> con_e ifelprefix ifprefix doprefix enum_head str_head
               name_lp

%type <intval> e .e term elist

%type <nodep>  atributos oatributos type enum_dcl struct_dcl cast_type
               null_decl funct_idn declarador fdeclarador nfdeclarador

%token <intval> CLASS NAME STRUCT RELOP CM DIVOP PLUS MINUS SHIFTOP MUL
                AND OR ER ANDAND OROR ASSIGN STROP INCOP UNOP ICON

%token <nodep> TYPE

/*  ------------------------- inicio da gramatica ---------------------- */

%%

def_ext_list:     def_ext_list def_externa
          |                                ={ }
          ;
def_externa:      data_def                 ={ }
          |  error                         ={ }
          |  ASM SM                        ={ }
          ;
data_def:
             oatributos  SM                                    ={ }
          |  oatributos inicio_dcl_list  SM                    ={ }
          |  oatributos fdeclarador { }  funcao_corpo          ={ }
          ;

funcao_corpo:     arg_dcl_list comporcmd
          ;
arg_dcl_list:     arg_dcl_list declaracao
          |                          ={ }
          ;

stmt_list:        stmt_list comando
          |                          ={ }
          ;

dcl_stat_list  :  dcl_stat_list atributos SM                   ={ }
          |  dcl_stat_list atributos inicio_dcl_list SM        ={ }
          |
          ;

declaracao: atributos declarador_list  SM          ={ }
          |  atributos SM                          ={ }
          |  error  SM                             ={ }
          ;
```

```
oatributos:       atributos
          |                          ={ }
          ;

atributos:  class type                    ={ }
          |  type class                   ={ }
          |  class                        ={ }
          |  type                         ={ }
          |  type class type              ={ }
          ;

class:            CLASS                   ={ }
          ;

type:          TYPE
          |  TYPE TYPE                    ={ }
          |  TYPE TYPE TYPE               ={ }
          |  struct_dcl
          |  enum_dcl
          ;

enum_dcl:      enum_head LC moe_list optcomma RC     ={ }
          |  ENUM NAME                                ={ }
          ;

enum_head:          ENUM                    ={ }
          |  ENUM NAME                      ={ }
          ;

moe_list:      moe
          |  moe_list CM moe
          ;

moe:           NAME                                 ={ }
          |  NAME ASSIGN con_e                      ={ }
          ;

struct_dcl:         str_head LC type_dcl_list optsemi RC ={ }
          |  STRUCT NAME                                   ={ }
          ;

str_head:      STRUCT                  ={ }
          |  STRUCT NAME                ={ }
          ;

type_dcl_list:     type_declaracao
          |  type_dcl_list SM type_declaracao
          ;

type_declaracao:  type declarador_list        ={ }
          |  type                                   ={ }
          ;
```

```
declarador_list:   declarador                              ={ }
          |  declarador_list  CM {}  declarador            ={ }
          ;

declarador:        fdeclarador
          |  nfdeclarador
          |  nfdeclarador COLON con_e %prec CM             ={ }
          |  COLON con_e %prec CM                          ={ }
          |  error                                         ={ }
          ;

nfdeclarador:      MUL nfdeclarador                        ={ }
          |  nfdeclarador  LP   RP                         ={ }
          |  nfdeclarador LB RB                            ={ }
          |  nfdeclarador LB con_e RB                      ={ }
          |  NAME                                          ={ }
          |   LP  nfdeclarador  RP                         ={ }
          ;

fdeclarador:       MUL fdeclarador                   ={ }
          |  fdeclarador  LP   RP                    ={ }
          |  fdeclarador LB RB                       ={ }
          |  fdeclarador LB con_e RB                 ={ }
          |  LP  fdeclarador  RP                     ={ }
          |  name_lp name_list  RP                   ={ }
          |  name_lp RP                              ={ }
          ;

name_lp:     NAME LP                         ={ }
          ;

name_list:         NAME                              ={ }
          |  name_list  CM  NAME                     ={ }
          |  error
          ;

inicio_dcl_list:        inicio_declarador %prec CM
          |  inicio_dcl_list  CM {}  inicio_declarador
          ;

xnfdeclarador:     nfdeclarador                      ={ }
          |  error
          ;

inicio_declarador:   nfdeclarador                                          ={ }
             |  fdeclarador                                                ={ }
             |  xnfdeclarador ASSIGN e %prec CM                            ={ }
             |  xnfdeclarador ASSIGN LC inicio_list optcomma RC            ={ }
             |  error
          ;

inicio_list:       inicializador %prec CM
           |       inicio_list  CM  inicializador
          ;
```

```
inicializador:     e %prec CM                          ={ }
              |  ibrace inicio_list optcomma RC        ={ }
              ;

optcomma  :
              |  CM
              ;

optsemi       :
              |  SM
              ;

optasgn       :                                       ={ }
              |  ASSIGN
              ;

ibrace        : LC                                    ={ }
              ;

comporcmd:          begin dcl_stat_list stmt_list RC
                   ={ }
              ;

begin:               LC
                   ={   }
              ;

comando:        e    SM
                   ={ }
              |  ASM SM { }
              |  comporcmd
              |  ifprefix comando
                   ={ }
              |  ifelprefix comando
                   ={ }
              |  doprefix comando WHILE  LP  e  RP   SM
                   ={ }
              |  BREAK  SM
                   ={ }
              |  CONTINUE  SM
                   ={ }
              |  RETURN  SM
                   ={ }
              |  RETURN e  SM
                   ={ }
              |   SM
              |  error  SM
              |  error  RC
              ;

doprefix: DO
                   ={ }
              ;
```

```
ifprefix: IF
                { }
        ;

ifelprefix:       ifprefix  comando ELSE
                ={ }
        ;

/*---------------------- Expressões ------------------------------------*/

con_e:              {  } e %prec CM   ={ }       ;
.e:         e
        |                             ={ }       ;
elist:              e %prec CM
        |   elist  CM  e              ={ }       ;

e:          e RELOP e                       ={ }
        |   e DIVOP e                       ={ }
        |   e  PLUS e                       ={ }
        |   e MINUS e                       ={ }
        |   e SHIFTOP e                     ={ }
        |   e MUL e                         ={ }
        |   e EQUOP  e                      ={ }
        |   e AND e                         ={ }
        |   e OR e                          ={ }
        |   e ER e                          ={ }
        |   e ANDAND e                      ={ }
        |   e OROR e                        ={ }
        |   e MUL ASSIGN e                  ={ }
        |   e DIVOP ASSIGN e                ={ }
        |   e PLUS ASSIGN e                 ={ }
        |   e MINUS ASSIGN e                ={ }
        |   e SHIFTOP ASSIGN e              ={ }
        |   e AND ASSIGN e                  ={ }
        |   e OR ASSIGN e                   ={ }
        |   e ER ASSIGN e                   ={ }
        |   e QUEST e COLON e               ={ }
        |   e ASOP e                        ={ }
        |   e ASSIGN e                      ={ }
        |   term                            ={ }
        ;
                                                          /* Operador: */
term:       term INCOP                                ={ } /* ++ ou -- */
        |   MUL term                                  ={ } /*      *     */
        |   AND term                                  ={ } /*      &     */
        |   MINUS term                                ={ } /*      -     */
        |   UNOP term                                 ={ } /*   ! ou ~    */
        |   INCOP term                                ={ } /* ++ ou -- */
        |   SIZEOF term                               ={ }
        |   LP cast_type RP term  %prec INCOP         ={ }
        |   SIZEOF LP cast_type RP  %prec SIZEOF      ={ }
        |   term  LB e RB                             ={ }
        |   funct_idn  RP                             ={ }
```

```
            |  funct_idn elist  RP                    ={ }
            |  term STROP NAME                        ={ }
            |  NAME                                   ={ } /* -> ou . */
            |  ICON                                   ={ } /* Const. Inteira */
            |  FCON                                   ={ } /* Const. Real    */
            |  STRING                                 ={ } /* Const. Alfa    */
            |  LP  e  RP                              ={ }
            ;

cast_type:      type null_decl                  ={ }
            ;

null_decl:       /* empty */                    ={ }
            |  LP RP                            ={ }
            |  LP null_decl RP LP RP            ={ }
            |  MUL null_decl                    ={ }
            |  null_decl LB RB                  ={ }
            |  null_decl LB con_e RB            ={ }
            |  LP null_decl RP                  ={ }
            ;

funct_idn:      NAME  LP                        ={ }
            |  term  LP                         ={ }
            ;

%%
```

Apêndice B

Especificação do Tipo Abstrato de Dados Arvore "AND-OR"

Este apêndice descreve a especificação formal por traço[1] do tipo abstrato de dados Arvore "And-Or". Esta árvore é utilizada para retratar o fluxo de controle e de dados de um código fonte. Os nós desta árvore podem ser do tipo "And", "Or", "Loop", "Call" e "Nill", representando, respectivamente, um bloco de programa, os comandos de decisão, as iterações, as chamadas às sub-routinas e os comandos de atribuição. Cada nó armazena uma série de informações necessárias para analisar e descrever o comando ou construtor representado.

Especificação Formal da Arvore "And-OR"

Esta especificação formal, que utiliza o método de traço, foi desenvolvida por [EVAN-90b] para o tipo abstrato Arvore "And-Or".

Notação matemática utilizada :

```
::    : equivalente a
^     : operador lógico e
v     : operador lógico ou
=     : igual (resultado)
==    : igual (comparação)
!=    : diferente
->    : retorna (resultado)
=>    : implica que
<=>   : se e somente se
```

[1] Método para especificação formal e abstrata de um software, permitindo que se descreva o comportamento das operações associadas a um tipo abstrato de dados [GART-77].

Tipo abstrato Arvore "And-Or"

Sintaxe :

O_funções :

A_CriaNull	(Andtree,lpar)	->	(Andtree,A)
A_Cria	(Andtree,nó,tipo,e)	->	(Andtree,nó)
A_Remove	(Andtree,nó)	->	(Andtree)
A_Destrói	(Andtree,A)	->	(Andtree)
A_Subárvore	(Andtree,A)	->	(Andtree)

V_funções :

A_Raiz	(Andtree,A)	->	(nó)
A_Pai	(Andtree,nó)	->	(nó)
A_Filho	(Andtree,nó,i)	->	(nó)
A_QtdFilho	(Andtree,nó)	->	(inteiro)
A_Busca	(Andtree,A,l)	->	(nó)
A_NFilho	(Andtree,nó)	->	(inteiro)
A_Antecessor	(Andtree,A,nó)	->	(nó)
A_Sucessor	(Andtree,A,nó)	->	(nó)
A_Tipo	(Andtree,nó)	->	(tipo)
A_Elemento	(Andtree,nó)	->	(e)
A_UltDesc	(Andtree,nó)	->	(nó)

onde :

Andtree	: Tipo abstrato ANDTREE
A	: Instância do tipo abstrato Andtree
no1,no2	: Nós da Andtree
e	: Conteúdo de um nó
l	: Número da linha do comando
i	: Número do filho
tipo	: Tipo do nó
lpar	: Lista de parâmetros da raiz da ANDTREE

Legalidade :

1) L(A_CriaNull(lpar))
2) L(T.A_Cria(no1,tipo,e)) => L(T.A_Cria(no1,tipo,e).A_Remove(no1))
3) L(T.Destroi(A))
4) (V(T.A_Tipo(no1) == AND) => L(T.A_Cria(no1,tipo,e))
5) ((V(T.A_Tipo(no1) != NILL) ^ (V(T.A_Tipo(no1) != CALL)) v
   ((V(T.A_Tipo(no1) == OR) ^ (V(T.A_QtdFilho(no1) < 2)) v
   ((V(T.A_Tipo(no1) == LOOP) ^ (V(T.A_QtdFilho(no1) == 0)) =>
   L(T.A_Cria(no1,tipo,e))

Equivalência :

1) V(T.A_CriaNull(lpar) == A) => T.A_CriaNull(lpar).A_Destroi(A) :: T
2) L(T.A_Cria(no1,tipo,e)) => T.A_Cria(no1,tipo,e).A_Remove(no1)) :: T
3) T.A_Raiz(A) :: T.A_Pai(no1) :: T.A_Filho(no1,i) ::
   T.A_QtdFilho(no1) :: T.A_NFilho(no) :: T.A_Antecessor(A,no) ::
   T.A_Sucessor(A,no) :: T.A_Tipo(no) :: T.A_Elemento(no) ::
   T.A_UltDesc(no) :: T.A_Busca(A,l) :: T

Valores :

1) V(A_CriaNull(lpar).T.A_Tipo(A_Raiz(A)) = AND
2) L(T.A_Cria(no1,tipo,e)) =>
   V(T.A_Cria(no1,AND,e).A_Tipo(no1) = AND
3) L(T.A_Cria(no1,tipo,e)) =>
   V(T.A_Cria(no1,OR,e).A_Tipo(no1) = OR
4) L(T.A_Cria(no1,tipo,e)) =>
   V(T.A_Cria(no1,LOOP,e).A_Tipo(no1) = LOOP
5) L(T.A_Cria(no1,tipo,e)) =>
   V(T.A_Cria(no1,CALL,e).A_Tipo(no1) = CALL
6) L(T.A_Cria(no1,tipo,e)) =>
   V(T.A_Cria(no1,NILL,e).A_Tipo(no1) = NILL
7) V(T.A_Busca(A,l)) = no1 <=> V(T.A_Elemento(no1)) = e,
   onde e é o conteúdo da linha l
8) V(T.A_NoFilho(no1,i)) = no2 <=> V(T.A_NFilho(no2)) = i
9) L(T.A_Cria(no1,tipo,e)) =>
   ((V(T.A_Cria(no1,tipo,e).A_Filho(no1,A_QtdFilho(no1)))) = no2)
    ^ (V(T.A_Pai(no2)) = no1)
10) L(T.A_Cria(no1,tipo,e)) =>
    V(T.A_Cria(no1,tipo,e).A_QtdFilho(no1)) = 1 + V(T.A_QtdFilho(no1))

Apêndice C

Algoritmos Referenciados no Capítulo 4

Este apêndice apresenta o detalhamento dos passos necessários à execução de alguns dos algoritmos referenciados no Capítulo 4. Vale ressaltar, que alguns deles se encontram especificados em "Program Desing Language - PDL", outros são descritos em uma forma mais informal.

Algoritmo C.1: Absorção do Operador "NOT"

Entrada : Tabela T, que representa um comando condicional em sua forma original.

Saída : Tabela T com o "NOT" absorvido.

Descrição : Este algoritmo descreve um método de absorção do operador "NOT", sendo utilizado no subsistema forma intermediária. Os seguintes casos serão considerados neste método de absorção:

1. (!operando1 op operando2)
2. (operando1 op !operando2)
3. (!operando1 op !operando2)
4. !(operando1 op operando2)
5. !(!operando1 op operando2)
6. !(operando1 op !operando2)
7. !(!operando1 op !operando2)

A coluna quatro de T conterá um número de 1 até 7, indicando uma das situações acima. O número -1 indica ausência do NOT.

Propriedades utilizadas:

NOT(A AND B) = NOT(A) OR NOT(B)
NOT(A OR B) = NOT(A) AND NOT(B)

Regra 1: Substituições de operadores

Operador	Novo Operador
AND	OR
≤	>
<	≥
≥	<
>	≤
=	≠
≠	=

Referência : Algoritmo formulado pelo autor da pesquisa.

Método:

1. FOR = Número de Linhas de T TO 1 DO

IF linha i de T é: (!Operando1 op Operando2) THEN

. Chamar o procedimento Absorve1, passando linha i e Operando1.

ENDIF;

IF linha i de T é: (Operando1 op !Operando2) THEN

. Chamar o procedimento Absorve1, passando linha i e Operando2.

ENDIF;

IF linha i de T é: (!Operando1 op !Operando2) THEN

. Chamar o procedimento Absorve1, passando linha i e Operando1.

. Chamar o procedimento Absorve1, passando linha i e Operando2.

ENDIF;

IF linha i de T é: !(Operando1 op Operando2) THEN

. Trocar operador (op) da linha i, utilizando a regra 1.

. IF Operando1 da linha i é Tabela
THEN Chamar Absorve1, passando linha i e Operando1.
ENDIF;

```
        . IF Operando2 da linha i é Tabela
          THEN Chamar Absorve1, passando linha i e Operando2.
          ENDIF;

    ENDIF;

    IF linha i de T é: !(!Operando1 op  Operando2) THEN

        . Trocar operador (op) da linha i, utilizando a regra 1.

        . IF Operando2 da linha i é Tabela
          THEN Chamar Absorve1, passando linha i e Operando2.
          ENDIF;

    ENDIF;

    IF linha i de T é: !( Operando1 op !Operando2) THEN

        . Trocar operador (op) da linha i, utilizando a regra 1.

        . IF Operando1 da linha i é Tabela
          THEN Chamar Absorve1, passando linha i e Operando1.
          ENDIF;

    ENDIF;

    IF linha i de T é: !(!Operando1 op !Operando2) THEN

        . Trocar operador (op) da linha i, utilizando a regra 1.

    ENDIF;

ENDFOR;
```

Algoritmo C1.1: Absorve1

Entrada : Operando e linha i de T

Saída : "NOT" absorvido no operando

Descrição : Procedimento recursivo que absorve o operador "NOT" no operando.

Referência : Algoritmo formulado pelo autor da pesquisa.

Método :

```
1. IF Operando é (Constante OR Variável OR Função)
   THEN
```

IF O outro Operando da linha i é vazio
THEN

. Mudar formato da linha:
  De: (Operando) Para: (Operando = 0)

ELSE

. Criar nova linha na Tabela T.
. Inserir a expressão (Operando = 0), como conteúdo da nova linha.

ENDIF;
ELSE /* Neste caso, operando aponta para uma linha da Tabela */
. Seja j a linha apontada pelo Operando.

IF Operador da linha j é relacional AND
   (Operando1 da linha j é (Variável OR Constante)) AND
   (Operando2 da linha j é (Variável OR Constante))
THEN

. Trocar operador da linha j de acordo com a regra 1.

ENDIF

IF Operador da linha j é ("And" OR "Or")
THEN

. Trocar operador da linha j, segundo a regra 1.

. Chamar recursivamente Absorve1, passando Operando1 e linha j como parâmetros.

. Chamar recursivamente Absorve1, passando Operando2 e linha j como parâmetros.

ENDIF
ENDIF

Algoritmo C.2: Forma Normal Conjuntiva (FNC)

Entrada : Tabela T, na forma original e com "NOT" absorvido.

Saída : Tabela T, em FNC.

Descrição : Gera a forma normal conjuntiva de expressões lógicas apresentadas numa tabela, obtendo-se três conjuntos: igualdade (I), outras relações (O) e disjunções (D).

Notação : Letras maiúsculas (X, Y, Z, etc) simbolizam expressões relacionais da forma (Operando1 op Operando2) ou apenas um boleano.

Propriedades das expressões boleanas utilizadas:

. "And" e "Or" são simétricos:

X ^ Y = Y ^ X
X ¦ Y = Y ¦ X

. "And" e "Or" são associativos:

X ^ (Y ^ Z) = (X ^ Y) ^ Z = X ^ Y ^ Z
(X ¦ Y) ¦ Z = X ¦ (Y ¦ Z) = X ¦ Y ¦ Z

. "Or" é distribuivel sobre "And":

(X ^ Y) ¦ Z = (X ¦ Z) ^ (Y ¦ Z)

Referência : Algoritmo formulado pelo autor da pesquisa.

Método :

1. CASE sobre os conectores ("And" e "Or") da tabela T.

1.1 CASE 1: Se todos conectores são "And", isto é, X AND Y AND ...

. Eliminar todos os conectores "And".

WHILE existir expressão X, Y, Z, etc
DO IF operador é igual THEN

. Inserir expressão no conjunto I.

ELSE

. Inserir expressão no conjunto O.

ENDIF;
ENDWHILE;

1.2 CASE 2: Se todos os conectores são "Or", isto é, X OR Y OR ...

. Eliminar todos os conectores "Or".

WHILE existir expressão X, Y, Z, etc
DO

. Inserir cada expressão no conjunto D, garantindo, porém, que as igualdades estejam juntas e em primeiro lugar.

ENDWHILE;

1.3 CASE 3: Se todos os conectores são "And", porém, existindo disjunções conectadas por "And", isto é:
X AND Y AND (W OR Z OR ...) AND ...

1.3.1 WHILE existir expressão, ei, da forma (W OR Z OR ...)
DO

. Criar conjunto Di para expressão ei.

WHILE existir subexpressão de ei, ou seja, W, Z, etc.
DO

. Eliminar os conectores "Or".

. Inserir cada subexpressão (W, Z, etc) no conjunto Di correspondente, garantindo, todavia, que as igualdades estejam juntas e em primeiro lugar.

ENDWHILE;
ENDWHILE;

1.3.2 WHILE existir expressões, ei, conectadas por "And", ou seja, X AND Y AND ....
DO

. Eliminar todos os conectores "And".

IF operador é igual THEN

. Inserir expressão no conjunto I.

ELSE

. Inserir expressão no conjunto O.

ENDIF;
ENDWHILE;

1.4 CASE 4: Caso misto. Os conectores "Or" e "And" estão mesclados em vários níveis, por exemplo:
X OR Y OR (W AND Z) OR R AND (S OR Q) ...

1.4.1 WHILE existir grupos de disjunções, Di.
DO

. Rotular o grupo Di com um outro simbolo, ou seja, Di = (W OR Z) ---> W1, renomeando este grupo Di para W1.

ENDWHILE;

1.4.2 Colocar cada disjunção simples ou renomeada no conjunto V.

1.4.3 Eliminar todas as disjunções simples ou renomeadas.

1.4.4 FOR cada combinação (Ci) formada a partir dos elementos das conjunções (cada combinação possuirá um elemento de cada conjunção)
DO

. Criar um conjunto Di para armazenar disjunções.

. Concatenar o conjunto V com Ci e inserir o resultado em Di.

ENDFOR;

1.4.5 Expandir as expressões renomeadas em 1.4.1.

Exemplo: Para melhor compreender o algoritmo C.2 em relação ao caso 4.

Considere o seguinte comando condicional:

X OR Y OR (W AND S) OR (Q AND R) OR O

Aplicando os passos 1.4.1, 1.4.2, 1.4.3, 1.4.4 e 1.4.5, tem-se:

Passo1.4.1: Agrupar disjunções e renomear.
X OR Y ==> X1

Passo1.4.2: Inserir as disjunções simples ou renomeadas em V.
V = {X1, O}

Passo1.4.4: Combinar os elementos das conjunções com V.
obtendo-se:
(V, W, Q) (V, W, R) (V, S, Q) (V, S, R).
Neste caso, cada combinação é uma disjunção.

Passo1.4.5: Expandir os resultados, produzindo como resultado final:

AND(OR(X, Y, O, W, Q), OR(X, Y, O, W, R)
OR(X, Y, O, S, Q), OR(X, Y, O, S, R))

Algoritmo C.3: SLICER

Entrada : Quádrupla SC={V, P, N1, N2}, na qual:
V = Conjunto das variáveis de interesse.
P = Programa representado pela "And-Or-Tree".
N1 = Primeira linha do programa na qual o processo irá terminar.
N2 = Ultima linha do programa na qual o processo irá começar.

Saída : Slice : Conjunto dos comandos que podem afetar o critério dado como entrada.
Var_Slice : Conjunto de variáveis que podem afetar o critério.

Descrição : Este algoritmo foi proposto por Sobrinho em 1984 (vide Capítulo 2) e possui como base uma Arvore "AND-OR" que representa o fluxo seqüencial de execução do programa. Deste modo, o algoritmo manipula esta árvore, percorrendo-a no sentido inverso (de baixo para cima), selecionando em cada nó da árvore as variáveis referenciadas e os comandos que afetam ou podem afetar as variáveis de interesse.

Notação :

1. SLICING : Aplicação do algoritmo SLICER sobre um determinado programa.

2. SLICER : Algoritmo para obtenção dos conjuntos SLICE e VAR_SLICE.

3. Para um dado comando y em P, temos :

   DEF(y) = {v / v é uma variável definida em y}
   REF(y) = {v / v é uma variável referenciada em y}
   PARAMR(y) = {v/ y é um comando do tipo CALL e v é uma variável passada como parâmetro em y}

4. CSET = Conjunto de variáveis que afetam as variáveis de interesse até um determinado momento do processo SLICING.

Referência : [REZE-92]

Método :

1. Obter os comandos inicial e final a partir das linhas inicial e final.

2. Marcar recursivamente os nós da ANDTREE a partir da função Current_Set();

2.1 CSET = V /* Inicialmente, o CSET é igual ao conjunto V que foi fornecido como entrada */

2.2 WHILE existir comando "m", a partir do comando final até o comando inicial

. Analisar as variáveis do comando, utilizando a sub-rotina recursiva Current_Set, passando "m" e CSET como parâmetros.

ENDWHILE;

3. Percorrer a árvore P a partir da sua raiz. Verificar a cada nó ("m") se ele foi marcado, caso afirmativo, deve-se:

3.1 Inserir m ao conjunto Slice

3.2 Inserir REF(m) ao conjunto Var_Slice

Algoritmo C.3.2: Current_Set

Entrada : m : Comando que está sendo analisado

CSET : Esta variável, no início do algoritmo, armazena o conjunto de variáveis de interesse.

Saída : CSET : Esta variável, no final do algoritmo, armazena todas as variáveis que podem afetar o critério até o comando ("m") em análise.

Descrição : A sub-rotina Current_Set executa um determinado procedimento para cada tipo de nó da Arvore "And-Or". Este procedimento tem a função de marcar os nós da "And-Or", a partir do momento em que o nó afete o critério. As variáveis do nó marcado serão inseridas em CSET simultaneamente.

Notação : Marcador(i) : Indicador boleano para determinar que o nó afeta o CSET de entrada.
cset(m) : Retorna o CSET do nó "m".

Referência : [REZE-92]

Método :

1. CASE tipo do nó do comando m

1.1 CASE1: Se m é um nó NILL

IF Marcador(m) = 1 THEN

. cset(m) = CSET - DEF(m)

```
ELSE
   IF (CSET ∩ DEF(m) ≠ ∅) THEN

      . cset(m) = ( CSET - DEF(m) ) U REF(m)
      . Marcador(m) = 1

    ELSE

      . cset(m) = CSET

    ENDIF;
ENDIF;
```

1.2 CASE2: Se m é um nó CALL

```
IF Marcador(m) = 0 AND ( PARAMR(m) ∩ CSET ≠ ∅)
THEN

     . Marcador(m) = 1
     . cset(m) = CSET

ENDIF;
```

1.3 CASE3: Se m é um nó AND

```
. Flag = 0

FOR k = Quantidade de filhos de m TO 1
DO

   . n           = k-ésimo filho de m
   . CSET        = Current_Set(n,CSET)
   . Flag        = Flag ou Marcador(n)
   . cset(m)     = CSET
   . Marcador(m) = Flag

ENDFOR;
```

1.4 CASE4 : Se m é um nó OR

```
. k  = Quantidade de filhos de m
. n1 = O primeiro filho de m      /* Parte Then */
. Current_Set(n1,CSET)

IF (k = 2)                        /* Isto é, tem then e else */
THEN

     . n2   = Segundo filho de m
     . Current_Set(n2,CSET)
     . Aux  = cset(Filho(m,1)) U cset(Filho(m,2))
     . Flag = Marcador(n1) OR Marcador(n2)
```

```
ELSE

      . Aux  = cset(Filho(m,1)) U CSET
      . Flag = Marcador(n1)

ENDIF;

/* Analise da condição */

IF  (Aux ∩ DEF(m) ≠ ∅)
THEN

      . Marcador(m) = 1
      . cset(m)      = ( Aux - DEF(m) ) U REF(m)

ELSE

      . Marcador(m) = Flag

      IF ( Flag = 1)
      THEN

            . cset(m) = Aux U REF(m)

      ELSE

            . cset(m) = Aux

      ENDIF;
ENDIF;
```

1.5 CASE5: Se m é um nó LOOP

```
. Band = 1
. n = Primeiro filho de m     /* Este filho é um nó AND */

IF ( CSET ∩ DEF(m) ≠ ∅)
THEN

      . CSET = (CSET - DEF(m) ) U REF(m)
      . Marcador(m) = 1
      . Band = 0

ENDIF;

. DIF = Current_Set(n,CSET) - CSET
. Marcador(m) = Marcador(m) OR Marcador(n)

IF (Marcador(m) = 0)
THEN

      . cset(m) = CSET
ELSE
```

```
            IF (Band)
            THEN

                . CSET = CSET U REF(m)

            WHILE(DIF ≠ ∅)
            DO

                . CSET = CSET U DIF
                . DIF = Current_Set(n,CSET - DEF(m)) - CSET

            ENDWHILE;

            . cset(m) = CSET

        ENDIF;
    ENDCASE;
```

Algoritmo C.4: Bloco (Obtenção dos lideres de blocos)

Entrada : M : Um nó da Arvore "And-Or"

Saida : Lider : Conjunto que corresponde aos lideres de blocos de comandos.

Descrição : Este algoritmo se utiliza da característica da "And-Or" e do fato dela estar estruturada para determinar quais os nós que lideram um conjunto de comandos.

Para determinação dos líderes de toda árvore, a primeira chamada desta sub-rotina deverá passar M como a raiz da "And-Or".

Referência : Algoritmo formulado pelo autor da pesquisa.

Método :

```
1. IF (tipo do nó M é "And") OR (tipo do nó M é "Or") OR
      (tipo do nó M é "Loop")
   THEN
        . Inserir M no conjunto Lider

        FOR i= 1 TO número de filho de M
        DO

            . N = i-ésimo filho de M
            . Chamar esta sub-rotina (Bloco), recursivamente, passando
              N como parâmetro

        ENDFOR;
   ENDIF;
```

Algoritmo C.5: Obtenção dos Programas Primos

Entrada : P : Uma Arvore "And-Or"

Saída : Primo : Uma lista de nós que correspondem ao início de cada programa primo. A ordem dos elementos na lista indica a hierarquia dos primos em P.

Descrição : Invoca a sub-rotina Blocos para determinar os líderes, a partir dos quais será determinada a hierarquia de primos.

Referência : Algoritmo formulado pelo autor da pesquisa.

Método :

```
1. . N = Raiz da árvore P
   . Líder = Bloco(N)        /* Invocando o algoritmo C.4 */
   . Primo = { }             /* Conjunto Vazio */

   FOR i = Número de elementos de Líder TO 1
   DO

       . Primo = Primo U (i-ésimo elemento de Líder)

   ENDFOR
```

Apêndice D

Funções que Podem Ser Utilizadas para Síntese do "loop"

Este apêndice descreve uma série de funções, que podem ser utilizadas pelo usuário da Ferramenta de Abstração Funcional para sintetizar o efeito do "loop" com respeito a uma determinada variável definida em seu corpo. Vale ressaltar que as expressões listadas abaixo foram baseadas no trabalho de Cheatham [CHEA-79].

1) **Array(<j1, ..., jn>, <s1, ..., sn>, E(j1, ..., jn))**
   Esta função representa "array" de dimensão <s1 x s2 x ... x sn>, cujos elementos indexados por <j1, ..., jn> são as expressões E(j1, ..., jn). Por exemplo:
   . Array(<j1,j2>, <n,n>, 1) -> Array unitário de dimensão n por n.
   . Array(<j1,j2>, <n,n>, Case(j1=j2, 1, j1 ≠ j2,0)) -> Matriz identidade.

2) **Primeiro(j, I, F, p(j))**
   Esta função representa o primeiro j, tal que I ≤ j ≤ F e p(j) = verdadeiro. Caso todos p(j) sejam falsos, j = F + 1.

3) **Último(j, I, F, p(j))**
   Esta função representa o último j, tal que I ≤ j ≤ F e p(j) = verdadeiro. Caso todo p(j) = falso, então j = F + 1.

4) **Soma_finita(j, I, F, expr(j))**
   expr(I) + expr(I+1) + ... + expr(F)

5) **Produto_finito(j, I, F, expr(j))**
   expr(I) * expr(I+1) * ... * expr(F)

6) **Exponencial_Finita(j, I, F, expr(j))**
   expr(I) ^ expr(I+1) ^ ... ^ expr(F)

7) Lambda(j, h(j))

Esta expressão pode ser utilizada pelo usuário para indicar a funcionalidade do "loop". Neste caso, "lambda" é associada a uma função recursiva, h(j), com o objetivo de reproduzir o efeito do "loop".

Alguns comentários sobre as funções acima se fazem necessários. Em primeiro lugar, qualquer comando condicional que apareça dentro destas funções não gera dois caminhos diferentes no fluxo de controle. Por exemplo, a expressão array(<j1,j2>, <n,n>, case(j1=j2, 1, j1 ≠ j2, 0)) é fechada em relação a sua funcionalidade e aos seus efeitos sobre o fluxo de controle do programa, ou seja, o "case" interno não produz qualquer desvio condicional no fluxo do programa. Em segundo lugar, os predicados "p(j)" das expressões "Primeiro" e "Último" não são simplesmente cópias dos predicados encontrados no programa, mas associações entre a fórmula recorrente que afeta o predicado e a condição de parada. Por exemplo, considere o programa:

```
While(r > 0) {
     r = r / 2;
}.
```

Neste caso, a relação de recorrência é "$r_{k+1} = r_k / 2$" e a função de saída para o "loop" é Último(j, 1, limite, $r / 2^j > 0$).

Apêndice E

Arquivos Gerados pelo Subsistema "Gerador da Forma Intermediária"

O subsistema "Gerador da Forma Intermediária" é responsável pela geração da Arvore "And-Or" (forma intermediária) de um programa alvo escrito na linguagem "C". Neste sentido, dado um programa de entrada, este susbsistema da Ferramenta de Abstração Funcional (FAF) gera uma série de arquivos que registram as informações estáticas relevantes para que o programa possa ser segmentado, decomposto e sintetizado.

A figura E.1 ilustra um programa alvo a ser transformado para a forma intermediária. As figuras E.2 até E.5 retratam os arquivos gerados pela ferramenta "Andtree".

```
    extern double distance, time;
    extern int    error;

    void Dck(station, starship, thrust, velocity, deltat)
    double station, starship, thrust, velocity, deltat;
    {
 1  double gconst, gravity, constacc, curvel, nextvel;
 2
 3  if (station <= 0.0 || starship <= 0 || thrust <= 0 ||
        velocity <= 0 || deltat <= 0.0 || time <= 0.0 ||
        distance <= 0.0) {
 4      error = 1;
 5  }
 6  else {
 7        gconst = 6.67 * 1.e-11;
 8        gravity  = gconst * station * starship /(distance*distance);
 9        constacc = gravity - thrust / starship;
10        curvel = velocity;
11        nextvel = curvel + constacc * deltat;
12
13        do {
14             distance = distance - curvel * deltat;
15             curvel   = nextvel;
16             time     = time + deltat;
17             nextvel = curvel + constacc * deltat;
18        } while(nextvel > 0.0);
19        error = 0;
    }
```

Figura E.1: Programa "DOCKING".

```
DCK.C
0;0;;;;0;1
Dck;station#Dck,.;starship#Dck,.;thrust#Dck,.;velocity#Dck,.;deltat#Dck,.
0,0
0;0;0
1;5;station#Dck,starship#Dck,thrust#Dck,velocity#Dck,deltat#Dck,time#G,distance#G;;;0;1
6;5;9;11;12;13;14;17;18;19;20;16;23;8
-1,0
0;0;1;7
0;5;;;5;1;1
0,0
0;0;0
4;6;;error#G;5;0;0
100507,1
0;0;0
0;8;;;5;0;1
0,0
0;0;0
4;9;;gconst#Dck;5;1;0
100508,2
0;0;0
4;11;gconst#Dck,station#Dck,starship#Dck,distance#G;gravity#Dck;5;1;0
100509,3
0;0;0
4;12;thrust#Dck,starship#Dck,gravity#Dck;constacc#Dck;5;1;0
100510,4
0;0;0
4;13;velocity#Dck;curvel#Dck;5;1;0
100511,5
0;0;0
4;14;constacc#Dck,deltat#Dck,curvel#Dck;nextvel#Dck;5;1;0
100512,6
0;0;0
1;16;nextvel#Dck;;5;1;1
17;18;19;20;16
-1,7
0;1;0
0;16;;;16,5;0;1
0,0
0;0;0
4;17;curvel#Dck,deltat#Dck,distance#G;distance#G;16,5;1;0
100500,8
0;0;0
4;18;nextvel#Dck;curvel#Dck;16,5;1;0
100511,9
0;0;0
4;19;time#G,deltat#Dck;time#G;16,5;1;0
100501,10
0;0;0
4;20;constacc#Dck,deltat#Dck,curvel#Dck;nextvel#Dck;16,5;0;0
100512,11
0;0;0
4;23;;error#G;5;0;0
100507,12
0;0;0
```

Figura E.2: Arquivo DCK.ATF

```
13
distance#G 7
time#G 7
deltat#Dck 7
velocity#Dck 7
thrust#Dck 7
starship#Dck 7
station#Dck 7
error#G 4
gconst#Dck 7
gravity#Dck 7
constacc#Dck 7
curvel#Dck 7
nextvel#Dck 7
```

Figura E.3: Arquivo "DCK.VAR"

```
5
0.0$#
0$#
1$#
6.67$#
1.e-11$#
```

Figura E.4: Arquivo "DCK.CON"

```
13
0 7
3 100500 100000 -1 -1 -1
3 100501 100001 -1 -1 -1
3 100502 100002 -1 -1 -1
3 100503 100003 -1 -1 -1
3 100504 100004 -1 -1 -1
3 100505 100005 -1 -1 -1
3 100506 100006 -1 -1 -1
1 1
-1 100007 -3 -1 -1 -1
2 1
15 100009 100008 -1 -1 -1
3 4
14 302 301 -1 -1 -1
15 100500 100500 -1 -1 -1
15 303 100505 -1 -1 -1
15 100508 100506 -1 -1 -1
4 2
17 100509 401 -1 -1 -1
14 100504 100505 -1 -1 -1
5 1
-1 100503 -2 -1 -1 -1
6 2
16 100511 601 -1 -1 -1
15 100510 100502 -1 -1 -1
7 1
3 100512 100010 -1 3 2
8 2
17 100500 801 -1 -1 -1
15 100511 100502 -1 -1 -1
9 1
-1 100512 -2 -1 -1 -1
10 1
16 100501 100502 -1 -1 -1
11 2
16 100511 1101 -1 -1 -1
15 100510 100502 -1 -1 -1
12 1
-1 100011 -3 -1 -1 -1
```

Figura E.5: Arquivo "DCK.EQN"

O arquivo "Dck.atf" armazena todas as informações necessárias para se criar uma Arvore "And-Or", ou seja, possui informações acerca do fluxo de controle, do fluxo de dados, dos apontadores para a forma intermediária dos comandos do programa, etc. O arquivo "Dck.var" representa a tabela de símbolos do programa, onde cada linha possui o nome da variável, seu escopo

e seu tipo. O arquivo "Dck.con" armazena todas as constantes encontradas no programa. Finalmente, o arquivo "Dck.eqn" armazena a forma intermediária de todos os comandos de atribuição do programa, bem como os comandos condicionais nele encontrados.

Apêndice F

Exemplo Detalhado

Este apêndice descreve de forma detalhada o processo de abstração funcional do programa alvo "Docking" mostrado no apêndice E (figura E.1), o qual será analisado e sintetizado para as Variáveis de Estado "Time" e "Distance".

O programa alvo "Docking" tem por objetivo calcular o tempo e a distância percorrida por um navio até que sua velocidade seja reduzida a zero durante a aproximação de um cais.

A decomposição em primos do programa "Docking" deve ser seguida pela análise de cada primo, obedecendo a hierarquia dos mesmos. Neste caso, tem-se os seguintes programas primos:

Primo nível 4:

```
do {
     distance = distance - curvel * deltat;
     curvel   = nextvel;
     time     = time + deltat;
     nextvel = curvel + constacc * delat;
} while (nextvel > 0);
```

Primo nível 3:

```
gconst = 6.67 * 1.e-11;
gravity = gconst * station * starship / (distance * distance);
constacc = gravity - thrust / starship;
curvel = velocity;
nextvel = (curvel + constacc * deltat;
```
Síntese do primo nível 4;

Primo nível 2:

```
if (station <= 0.0 || starship <= 0 || thrust <= 0 ||
   velocity <= 0 || deltat <= 0.0 || time <= 0.0 ||
   distance <= 0.0)
```

Síntese do primo nível 3;

Primo nível 1:

Síntese do primo nível 2;

i) Síntese do primo nível 4

O primo nível 4 é um programa primo que representa um "loop", cujo corpo é formado apenas por comandos de atribuição.

i.a) Síntese do corpo do "loop"

O corpo do "loop" é formado apenas por seqüências simples. Sendo assim, basta verificar se alguma variável referenciada numa determinada expressão é definida em alguma outra expressão anterior dentro do primo. Caso afirmativo, a variável referenciada cede o seu lugar a um ponteiro que representa a expressão mais recente que a define, obtendo-se:

		Operador	Operando1	Operando2
Distance: ——>	3.1.1	-	Distance	E3.1.2
	2	*	Curvel	Deltat
Curvel : ——>	3.1.3	.	Nextvel	.
Time : ——>	3.1.4	+	Time	Deltat
NextVel : ——>	3.1.5	+	E3.1.3	E3.1.6
	6	*	Constacc	Deltat

A variável Curvel da expressão E3.1.5 é a única variável referenciada deste primo que se encontra definida anteriormente (equação E3.1.3), devendo ser substituída pelo seu ponteiro.

i.b) Síntese do construtor "loop"

A síntese do corpo do "loop" deve ser seguida pela sua abstração em uma forma fechada para cada variável modificada em seu corpo.

i.b.1) Slicer para cada variável definida no corpo do "loop"

O "Slicer" deve ser aplicado à cada variável definida no corpo do "loop", obtendo-se as seguintes relações de recorrência:

Distance: E3.1.1 e E3.1.3
Curvel: E3.1.3
Time: E3.1.4
Nextvel: E3.1.5 e E3.1.3

As relações anteriores devem ser reordenadas de tal maneira que as relações simples sejam sintetizadas prioritariamente:

1. Time : E3.1.4
   $Time_{k+1} = Time_k + Deltat$
2. Nextvel : E3.1.5
   $Nextvel_{k+1} = Nextvel_k + Constacc * Deltat$
3. Curvel : E3.13
   $Curvel_{k+1} = Nextvel_k$
4. Distance : E3.1.1 e E3.1.3
   $Distance_{k+1} = Distance_k - Curvel * Deltat$

i.b.2) Solução das relações de recorrência

A solução para cada relação de recorrência acima é:

1. Solução Simples:
   Time = $Time_0$ + Soma_finita(j, 1, IL, Deltat)
   Nextvel = $Nextvel_0$ + Soma_finita(j, 1, IL, Constacc * Deltat)

2. Solução composta, pois depende de resultados anteriores:

$$Curvel = Nextvel_o + Soma\_finita(j, 2, IL, Constacc * Deltat)$$
$$Distance = Distance_o - Soma\_finita(i, 1, IL, Nextvel_o + Soma\_Finita(j, 3, i, Constacc*Deltat)) * Deltat$$

Alguns comentários são necessários para melhor entendimento da resolução das relações de recorrência acima. As relações de recorrência simples podem ser facilmente resolvidas. As relações de recorrência dependentes são resolvidas em termos das soluções encontradas para as relações de recorrência simples. Por exemplo, a solução para relação de recorrência da variável "curvel" depende da solução encontrada para "nextvel", levando-se em conta a posição relativa de ambas relações no corpo do "loop". Como "curvel" encontra-se definida antes que "nextvel", a solução para "curvel" deve ser defasada de um ciclo do "loop", ou seja:

$$Nextvel = Nextvel_o + Soma\_finita(j, \mathbf{1}, IL, Constacc * Deltat)$$
$$Curvel = Nextvel_o + Soma\_finita(j, \mathbf{2}, IL, Constacc * Deltat)$$

A solução para variável "Distance" é um pouco mais complexa, pois, além de envolver a defasagem de mais um ciclo em relação à "Curvel", exige uma solução que envolve dois somatórios aninhados. Neste caso, o limite superior do somatório mais interno depende do valor do índice do somatório mais externo, isto é:

$$Curvel = Nextvel_o + Soma\_finita(j, \mathbf{2}, IL, Constacc * Deltat)$$
$$Distance = Distance_o - Soma\_finita(\mathbf{i}, 1, IL, Nextvel_o + Soma\_Finita(\mathbf{j}, \mathbf{3}, \mathbf{i}, Constacc*Deltat)) * Deltat$$

As tabelas de equações para as quatro relações de recorrência do corpo do "loop" são listadas abaixo:

			Operando1	Operando2
Time:	4.1	+	Time	Soma_Finita(j,1,IL,Deltat)
Nextvel:	4.2	-	Nextevel	Soma_Finita(j,1,IL,E4.3)
	3	*	Constacc	Deltat
Curvel:	4.4	+	Nextvel	Soma_Finita(j,2,IL,E4.3)
Distance:	4.5	-	Distance	Soma_Finita(i,1,IL,E4.6)
	6	*	E4.7	Deltat
	7	+	Nextvel	Soma_Finita(j,3,i,E4.8)
	8	*	Constacc	Deltat

3. Solução para IL (número de iterações):

IL = 1 + Floor[Abs (N)], onde:

$$N = \frac{Deltat * Constacc + Nextvel}{Deltat * Constacc}$$

ii) Síntese do primo nível 3

Este primo é formado por seqüências, permitindo que sua abstração seja produzida a partir de rearranjos dos ponteiros. Sendo assim, a abstração será obtida a partir da composição funcional de todos os nós do bloco "And" em questão, isto é:

			Operando1	Operando2
Gconst:	3.1	*	6.67	E3.2
	2	^	10	-11
Gravity:	3.3	*	E3.1	E3.4
	4	*	Station	E3.5
	5	/	Starship	E3.6
	6	^	Distance	2
Constacc:	3.7	-	E3.3	E3.8
	8	/	Thrust	Starship
Curvel:	3.9	.	Velocity	.
Nextvel:	3.10	+	E3.9	E3.11
	11	*	E3.7	Deltat
Time:	3.12	+	Time	Soma_Finita(j,1,IL,Deltat)
Nextvel:	3.13	+	E3.10	Soma_Finita(j,1,IL,E3.14)
	14	*	E3.7	Deltat
Curvel:	3.15	+	E3.10	Soma_Finita(j,2,IL,E3.14)
Distance:	3.16	-	Distance	Soma_Finita(i,1,IL,E3.17)
	17	*	E3.18	Deltat
	18	+	E3.10	Soma_Finita(j,3,i, E3.19)
	19	*	E3.7	Deltat

iii) Síntese do primo 2

Este primo representa uma decisão e requer a análise da sua parte "then" e "else", isto é:

Primeiro caso: Se o predicado for verdadeiro

Condição de Execução:

Ou	≤	Station	0.0	.
Ou	≤	Starship	0.0	.
Ou	≤	Thrust	0.0	.
Ou	≤	Velocity	0.0	.
Ou	≤	Delat	0.0	.
Ou	≤	Time	0.0	.
	≤	Distance	0.0	.

Variáveis:

		Operador	Operando1	Operando2
Time:	2.1	.	Time	.
Distance:	2.2	.	Distance	.

Esta tabela indica que as variáveis "Time" e "Distance" possuem os seus valores iniciais.

Segundo caso: Se o predicado for falso

Condição de Execução:

Ou	≤	Station	0.0	Not
Ou	≤	Starship	0.0	Not
Ou	≤	Thrust	0.0	Not
Ou	≤	Velocity	0.0	Not
Ou	≤	Delat	0.0	Not
Ou	≤	Time	0.0	Not
	≤	Distance	0.0	Not

Variáveis:

			Operando1	Operando2
Gconst:	2.1	*	6.67	E2.2
	2	^	10	-11
Gravity:	2.3	*	E2.1	E2.4
	4	*	Station	E2.5
	5	/	Starship	E2.6
	6	^	Distance	2
Constacc:	2.7	-	E2.3	E2.8
	8	/	Thrust	Starship
Curvel:	2.9	.	Velocity	.
Nextvel:	2.10	+	E2.9	E2.11
	11	*	E2.7	Deltat
Time:	2.12	+	Time	Soma_Finita(j,1,IL,Deltat)
Nextvel:	2.13	+	E2.10	Soma_Finita(j,1,IL,E2.14)
	14	*	E2.7	Deltat
Curvel:	2.15	+	E2.10	Soma_Finita(j,2,IL,E2.14)
Distance:	2.16	-	Distance	Soma_Finita(j,1,IL,E2.17)
	17	*	E2.18	Deltat
	18	+	E2.10	Soma_Finita(j,3,i,E2.19)
	19	*	E2.7	Deltat

iv) Síntese do primo nível 1

Este primo é formado por uma seqüência, sendo o resultado da sua síntese igual ao resultado obtido para a síntese do primo de nível 2.

v) Expandir resultado final

O resultado obtido para a análise do primo de nível hierárquico mais alto deve ser expandido com o objetivo de facilitar a sua interpretação. Neste caso, tem-se:

CASO 1: station $\leq$ 0.0 OR starship $\leq$ 0.0 OR thrust $\leq$ 0.0 OR
velocity $\leq$ 0.0 OR deltat $\leq$ 0.0 OR distance $\leq$ 0.0

time := time

distance := distance

CASO 2: station $>$ 0.0 AND starship $>$ 0.0 AND thrust $>$ 0.0 AND
velocity $>$ 0.0 AND deltat $>$ 0.0 AND distance $>$ 0.0

time = time + IL * deltat

$$\text{distance} = \text{distance} - \sum_{i=1}^{IL} [(\text{velocity} + \sum_{j=2}^{i} [(6.67*10^{-11} * \text{station} * \text{starship} / \text{distance}^2 - \text{thrust} / \text{starship}) * \text{delta}]) * \text{deltat}]$$

ONDE:

IL = 1 + Floor[Abs (N)]

$$N = \frac{\text{deltat} * \left[\frac{6.67 * 10^{-11} * \text{starship} * \text{station}}{\text{distance}^2} - \frac{\text{Thrust}}{\text{starship}} \right] + \text{velocity}}{\text{deltat} * \left[\frac{6.67 * 10^{-11} * \text{starship} * \text{station}}{\text{distance}^2} - \frac{\text{Thrust}}{\text{starship}} \right]}$$

Apêndice G

Abstração de Decisões em Funções que Produzam um Único Valor

Este apêndice detalha as principais interfaces da FAF responsáveis pela obtenção de abstrações que produzam um único valor para as decisões.

O modelo da FAF prevê a necessidade futura do reconhecimento e abstração, com um simples valor, de programas primos que representem decisões (Capítulo 3). Esta abstração tem o objetivo de reduzir o número de casos produzidos durante o processo de abstração e, em adição, aumentar a legibilidade da abstração final.

Devido à inexistência de uma solução automatizada para este reconhecimento e abstração, a FAF incorpora interfaces que permitem ao usuário definir uma abstração de mais alto nível para os contrutores de decisão.

O exemplo da figura G.1 é um simples programa, utilizado para demonstrar estas interfaces.

```
#include <stdio.h>

teste12(b)
int b;
{
1     int a, e, c, d;
2
3     a = (int) (5 / b);
4     b = 10;
5     c = 15;
6     d = 12;
7     e = a * 15;
8
9     if (e > c)
10          e = c;
11
12    b = (a + b / c + d ) * d * (10 + 5*a) - 5*a*b*c/d;
13    c = a + b + 1;
14    d = 10.0 + 5.0 * (e * e / (e*5.0));
15    printf("%d %d %d %d %d\n", a ,b, c, d, e);
 }
```

Figura G.1: Exemplo Utilizado para Demonstrar Interfaces da FAF

Durante o processo de análise e abstração do programa da figura G.1, a FAF pergunta ao usuário, via interface da figura G.2, se ele deseja sintetizar o primo de decisão (linhas 9 e 10 do programa) em uma abstração que produza um único valor. Em caso afirmativo, a FAF ativa um editor que objetiva receber a função abstração do primo em questão.

```
        Verifique se o primo em questão pode ser
      representado em nível mais alto de abstração        100.0 %

Condição:
          e > c

Atribuições Concorrentes:
          e := c


Tecle : S)im  N)ão  :
```

Figura G.2: Tela utilizada para mostrar um primo decisão e perguntar se o usuário deseja abstraí-lo em mais alto nível

Nesse editor, cuja interface está ilustrada na figura G.3, o usuário tem à sua disposição uma série de informações e facilidades, por exemplo:

HOME : Permite posicionar o cursor no início da linha.
END : Permite posicionar o cursor no fim linha.
DEL : Apagar um Caracter.
CTRL O : Abre uma linha.
CTRL X : Armazena a função e volta para o processo de abstração.
CTRL A : Mostra sintaxe válida para especificar a abstração em mais alto nível. A figura G.4 ilustra esta tela de auxílio.
CTRL D : Apaga uma Linha.
CTRL V : Permite a visualização do primo a ser abstraído (vide figura G.2).
CTRL Q : Permite que o usuário desista de fornecer a abstração.

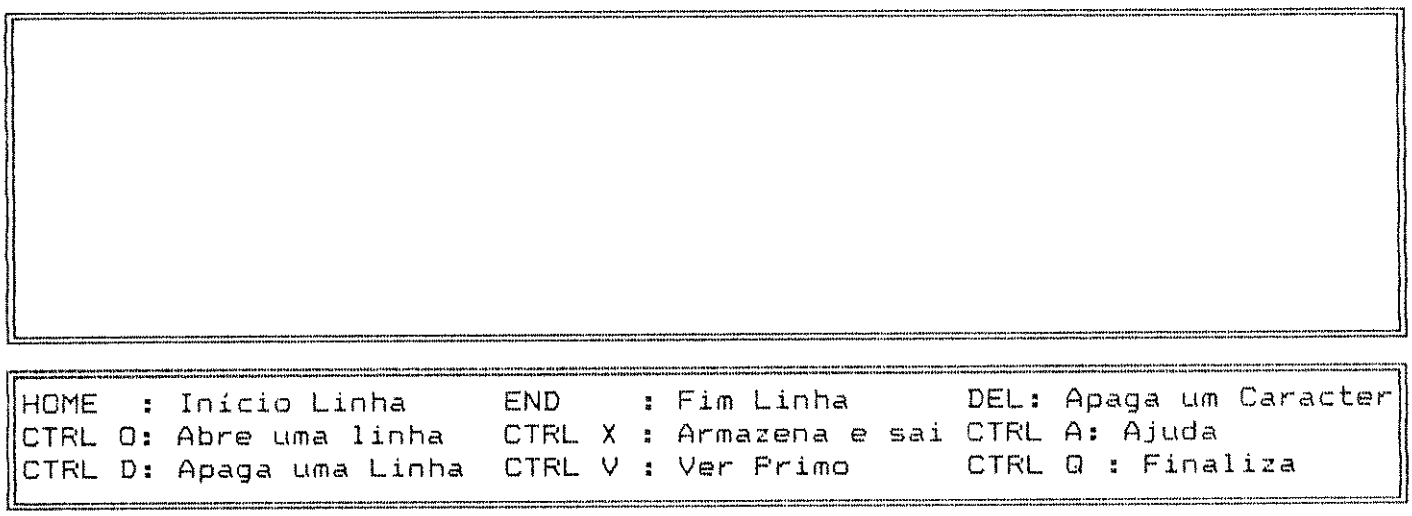


Figura G.3: Tela do Editor Utilizado para que o Usuário Informe a Função que Vai Abstrair o Primo Decisão em Questão

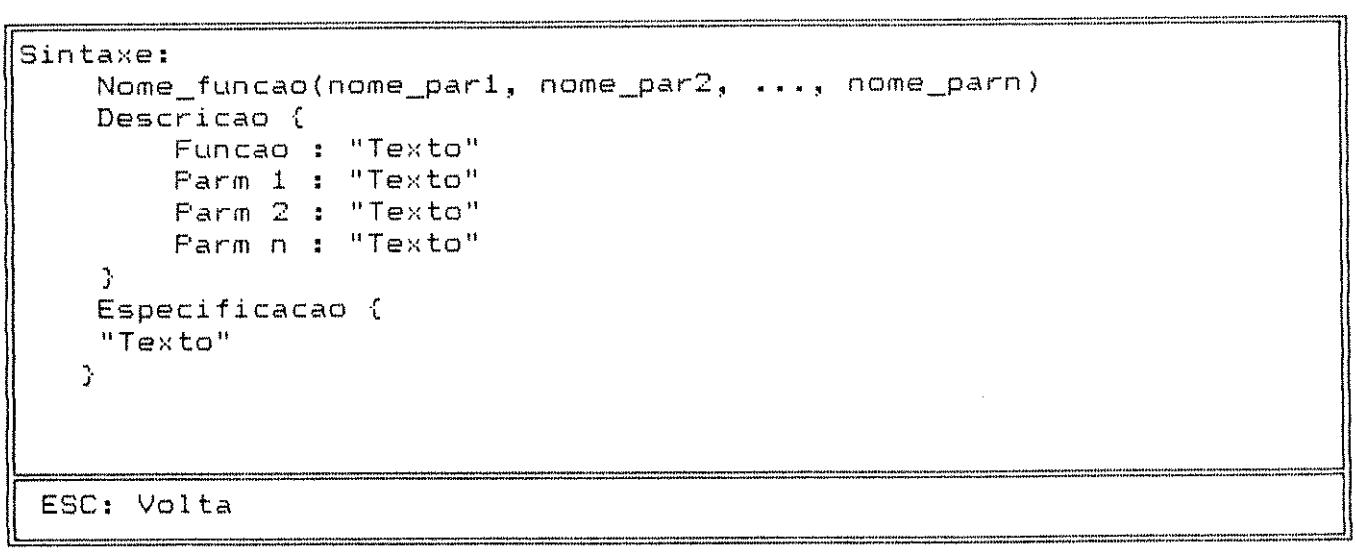


Figura G.4: Tela que informa a sintaxe válida para a função que vai abstrair um primo decisão

Voltando ao exemplo, pode-se verificar que o primo em questão:

```
if (e > c)
    e = c;
```

pode ser sintetizado em **minimo(e,c)**. Neste caso, o resultado final da abstração é ilustrado na figura G.5.

Atribuições Concorrentes:

```
a := 5/b

e := minimo(((5/b)*15), 15)

b := (((((5/b)+(10/15))+12)*12)*(10+(5*(5/b))))-
          ((((5*(5/b))*10)*15)/12)

c := ((5/b)+((((((5/b)+(10/15))+12)*12)*(10+(5*(5/b))))-
          ((((5*(5/b))*10)*15)/12)))+1

d := 10.0+(5.0*((minimo(((5/b)*15), 15)*minimo(((5/b)*15), 15))/
          (minimo(((5/b)*15), 15)*5.0)))
```

Figura G.5: Resultado final da abstração do programa da Figura G.1 (caso seja fornecida uma abstração que produza um único valor para o primo de tipo decisão do programa)

Observe que o resultado da figura G.5 possui apenas um possível caso de execução, apesar da existência do comando "if" no programa. Por outro lado, se o mesmo programa fosse abstraído pela FAF sem o fornecimento da função **minimo(e,c)**, o resultado final da abstração possuiria dois possíveis casos de execução (vide figura G.6).

```
---------------------------- Caso 1 -------------------------------

Condição de Execução:

     ((5/b)*15) > 15

Atribuições Concorrentes:

     a := 5/b

     e := 15

     b := (((((5/b)+(10/15))+12)*12)*(10+(5*(5/b))))-
          ((((5*(5/b))*10)*15)/12)

     c := ((5/b)+((((((5/b)+(10/15))+12)*12)*
          (10+(5*(5/b))))-((((5*(5/b))*10)*15)/12)))+1

     d := 10.0+(5.0*(((15)*(15))/((15)*5.0)))

---------------------------- Caso 2 -------------------------------

Condição de Execução:

     NOT(((5/b)*15) > 15)

Atribuições Concorrentes:

     a := 5/b

     e := (5/b)*15

     b := (((((5/b)+(10/15))+12)*12)*(10+(5*(5/b))))-
          ((((5*(5/b))*10)*15)/12)

     c := ((5/b)+((((((5/b)+(10/15))+12)*12)*(10+(5*(5/b))))-
          ((((5*(5/b))*10)*15)/12)))+1

     d := 10.0+(5.0*((((5/b)*15)*((5/b)*15))/(((5/b)*15)*5.0)))
```

Figura G.6: Resultado final da abstração do programa da Figura G.1 (caso não seja fornecida uma abstração que produza um único valor para o primo de tipo decisão do programa)

Glossário

Abstração Funcional:

Representa a funcionalidade de um programa alvo, a qual descreve o efeito do programa sobre suas variáveis em todas as possíveis situações.

Arquitetura de um Programa:

Representa as estruturas e o relacionamento entre os componentes de um programa [IEEE-83].

Arvore "And-Or":

Tipo abstrato de dados que representa o fluxo de dados e de controle do programa.

Casos de Execução:

Representa o resultado obtido pela Ferramenta de Abstração Funcional. Cada Caso é formado por uma condição (domínio) e uma série de expressões simbólicas, representando um possível efeito do programa sobre os seus dados.

Computações Independentes de um "loop":

Conjunto de computações de um "loop", formado por todos os comandos e variáveis que não afetam quaisquer outras computações do corpo do "loop" em questão [WALT-79].

Conjunção de Condições:

Indica a união lógica entre duas ou mais expressões condicionais.

Construtor para Decisão:

Construtor das linguagens de programação utilizado para controlar a execução de um bloco de comandos do programa, dependendo de alguma expressão condicional. Este construtor é representado pelos comandos "if e if-then-else".

Construtor para Iteração:

Construtor das linguagens de programação utilizado para que um conjunto de instruções seja executado até que ocorra uma determinada condição de término. Este construtor é representado pelos comandos "do-until", "while" e "for".

Construtor para Seqüência:

Construtor das linguagens de programação utilizado para representar um bloco de comandos, os quais devem ser executados incondicionalmente e em ordem.

Critério do "Slicer":

Especifica o comportamento do programa que se deseja extrair com a técnica de "Slicer". O critério é constituido por um conjunto de variáveis e uma linha de código.

Execução Simbólica:

Técnica de verificação e análise em que a execução do programa é simulada. Nesta simulação, utilizam-se símbolos no lugar dos dados de entrada e as saídas geradas pelo programa são expressas em termos de expressões lógicas ou matemáticas [IEEE-83].

Expressão Condicional:

Condição que controla os comandos "if e if-then-else" e "do-until". Esta condição é formada por várias expressões relacionais que, por sua vez, são agregadas através dos operadores boleanos "And", "Or" e "Not".

Ferramenta de Abstração Funcional (FAF):

Programa de computador utilizado para auxiliar o entendimento de outros programas.

Filtro de um "loop":

Computação que restringe uma seqüência de valores de entrada [WALT-79], podendo ser composta com as computações independentes para determinar quais os valores que estarão disponíveis para o processamento em questão.

Função Programa:

Propósito específico de um programa ou parte dele.

"Loop" Básico:

Formado pelos comandos que afetam o seu controle computacional, isto é, a iniciação, o teste de terminação e o comando que modifica a variável referenciada no teste.

Programa:

Seqüência de instruções apropriadas para processamento através de computadores. Este processamento pode incluir o uso de um compilador, interpretador ou tradutor com o objetivo de preparar o programa para execução [IEEE-83].

Programa Alvo:

Programa de computador, objeto de análise e síntese neste trabalho.

Programa Estruturado:

Programa construído apenas com os construtores para seqüência, decisão ou iteração. Cada bloco do programa deve ter apenas um ponto de entrada e um ponto de saída.

Programa Primo:

Todo programa próprio que não possua nenhum outro subprograma próprio, exceto ele mesmo [LING-79].

Programa Próprio:

Programa que possui um diagrama de fluxo com apenas um ponto de entrada e um ponto de saída, existindo um caminho a partir da entrada até a saída para todo e qualquer nó considerado [LING-79]

Reengenharia de Dados:

Método utilizado para reduzir o escopo das variáveis de um programa.

Relações de Recorrência:

Relações que envolvem fórmulas computacionais recorrentes em "loops". Nestas fórmulas, o valor de uma variável, em um determinado ciclo "k" de um "loop", pode depender do valor assumido pela mesma variável ou por outras variáveis no ciclo "k-1".

Síntese:

Indica a composição de elementos concretos ou abstratos em um todo. Os termos síntese e abstração possuem o mesmo significado neste trabalho.

"Slice":

Conjunto de comandos selecionados pela aplicação da técnica de "Slicer".

"Slicer":

Técnica que seleciona os comandos e/ou variáveis de um programa, as quais afetam direta ou indiretamente um conjunto de variáveis a partir de um comando de interesse [SOBR-84].

"Slicing":

Aplicação do algoritmo sobre um determinado programa.

Variáveis de Estado:

Conjunto de todas as variáveis que armazenam a intençao computacional do programa, sendo formado por todas as variáveis globais que recebam valor no programa, pelos parâmetros formais passados por referência que recebam valor, pelas variáveis de retorno do programa (comando "return" na linguagem C) e, finalmente, pelas variáveis referenciadas em comandos de gravaçao ou impressão.

"Trace-Table":

Tabela de equações, na qual cada linha corresponde a um comando seqüencial do programa e cada coluna corresponde a uma variável assinalada nos comandos que se encontram nas linhas da tabela. Cada célula da tabela indica o estado atual da variável da coluna em relação ao comando da linha.

"Trace-Table" Condicional:

"Trace-Table" para cada possível combinaçao das condições envolvidas nas atribuições condicionais.

YACC:

Utilitário genérico utilizado para descrever uma especificaçao de uma linguagem. A partir desta especificaçao, o YACC gera uma sub-rotina em "C" que reconhece e executa as ações pertinentes à especificaçao implementada [STEP-92].