Dissertação de Mestrado

Detecção de Paralelismo a partir de Semântica Denotacional e de Grafos de Dependências

Aluno: Carlos Bazílio Martins

Orientador: Edward Hermann Haeusler

A minha mãe Maria Piedade, a minha irmã Cláudia
e a todos os familiares, biológicos e
de consideração, que contribuíram
para este trabalho.

Agradecimentos

Primeiramente, gostaria de agradecer minha querida mãe. Mesmo se tratando de uma mãe, ela é um exagero. Capaz de abrir mão de sua vida pelo sucesso de sua prole.

Minha família (de Petrópolis) também teve participação fundamental. Tudo bem que a pressão contrária também era grande, afinal, sentíamos falta da convivência. Mas tudo isso faz parte.

Quanto ao fato de ter me referido a família de Petrópolis, isto se tornou comum depois destes mais de 8 anos fora de Petrópolis. Isto porque estive durante este longo tempo em contato com pessoas, famílias que me fizeram, fazem e espero que sempre me façam muito feliz. Muito obrigado a todos vocês.

Aos amigos do LMF, que me deram todo tipo de apoio: emocional, técnico e até alimentar :).

À turma *.93, minha querida turma da graduação. Obrigado por tudo: churrascos, campanhas para arrecar fundos, confidências, alegrias, tristezas e tudo mais que contribuiu para minha formação pessoal e profissional.

Ao orientador Hermann que, como havia dito durante a defesa, abdicou de uma pesquisa mais teórica, que é sua área de maior interesse, para que eu pudesse trabalhar com assuntos que mais me cativavam.

Finalmente, agradeço a todos que direta ou indiretamente ajudaram, não só na confecção deste trabalho, como também na formação do Bazílio, Baz, Baze, Bazilon 5, Bazildo, Bãzico, Nem, Negão, Chocolate, Bombom, Buiu, Baby, Carlinhos, ...

Muito obrigado mesmo a todos vocês

Carlos Bazílio Martins, 12/05/2000

Resumo

Este trabalho visa a criação de uma arquitetura genérica que tem como objetivo a geração automática de código paralelo a partir de programas sequenciais. Este tipo paralelismo é conhecido na literatura como paralelismo implicito. Inicialmente, a arquitetura recebe como entrada um programa e a especificação da linguagem em semântica denatocional referente a este, e retorna o código paralelizado, no mesmo nível de abstração do código de entrada. O paralelismo obtido é indicado através de construções paralelas do tipo *cobegin/coend* e *forall* num código intermediário que, posteriormente, é pós-processado para instanciação em uma arquitetura específica. Num protótipo apresentado, que processa a linguagem C, geramos código *multithreaded*.

Abstract

This work proposes a generic architecture that generates parallel automatic code from sequential programs. In literature this subject is called implicit parallelism. The architecture has a target language's denotational semantic specification and an example code in such a language as its input. The output is a parallel code in the same abstraction level as the original sequential code. Intermediate code generation with parallel constructs (cobegin/coend, forall, ...) is one of the intermediate steps. This code is processed to be instantiated to a particular machine. Our implemented prototype processes the C language and generates multithreaded programs from sequential ones.

Índice

Índice de Figuras

Capítulo 1 - Introdução

1.1 - Motivação

Devido ao rápido desenvolvimento de componentes de "hardware", a Engenharia de Software passou a ter mais um fator de fundamental importância na elaboração de um bom sistema: o aproveitamento eficiente dos recursos tecnológicos disponíveis.

Desta maneira, pesquisadores passaram a projetar ferramentas que teriam a preocupação de utilizar de forma inteligente todo "hardware" oferecido. Com isto, ramos da pesquisa em Computação como Paralelismo a Nível de Instrução – *ILP*, Sistemas Distribuídos e Paralelos, Escalonamento de Processos em Sistemas Operacionais, …, foram revigorados na tentativa de se diminuir o abismo de desempenho existente entre "hardware" e "software".

Dentre as abordagens citadas, decidimos trabalhar com linguagens de programação para ambientes paralelos, por ser tratar de uma área que acreditamos ainda ter muito a contribuir.

Em computação paralela, a tradicional visão de *Von Neumann* de um computador, onde apenas um único fluxo de instrução é executado por vez, é substituída por uma visão cooperativa, onde várias unidades processadoras trabalham em conjunto na resolução de um problema.

Com respeito a linguagens para máquinas paralelas, várias abordagens tem sido tratadas, dentre as quais podemos citar: programação de dados paralela, troca de mensagens, programação funcional …

Observe que, em todas as abordagens citadas, a responsabilidade principal fica a cargo do programador. Costuma-se dizer, nestes casos, que o paralelismo é explícito.

Dependendo das escolhas feitas por este programador, o resultado obtido com a paralelização pode ficar aquém do esperado.

Pensando desta forma, fomos motivados ao projeto de uma arquitetura onde o usuário não precisará se preocupar em como seu programa será paralelizado, em como seus dados deverão ser distribuídos para uma melhor computação. Esta forma de abordar o paralelismo também é conhecido como Paralelismo Implícito.

Em nossa arquitetura, o tipo de Paralelismo utilizado se aproxima da Programação Funcional. A causa disto é a definição da linguagem a ser tratada em Semântica Denotacional modificada, com a intenção de exprimir relações de dependência entre os termos sintáticos da linguagem.

1.2 - Objetivo

Este trabalho visa a criação de uma arquitetura genérica que tem como objetivo a geração automática de código paralelo a partir de programas seqüenciais. Um esquema geral da arquitetura pode ser visualizado abaixo:

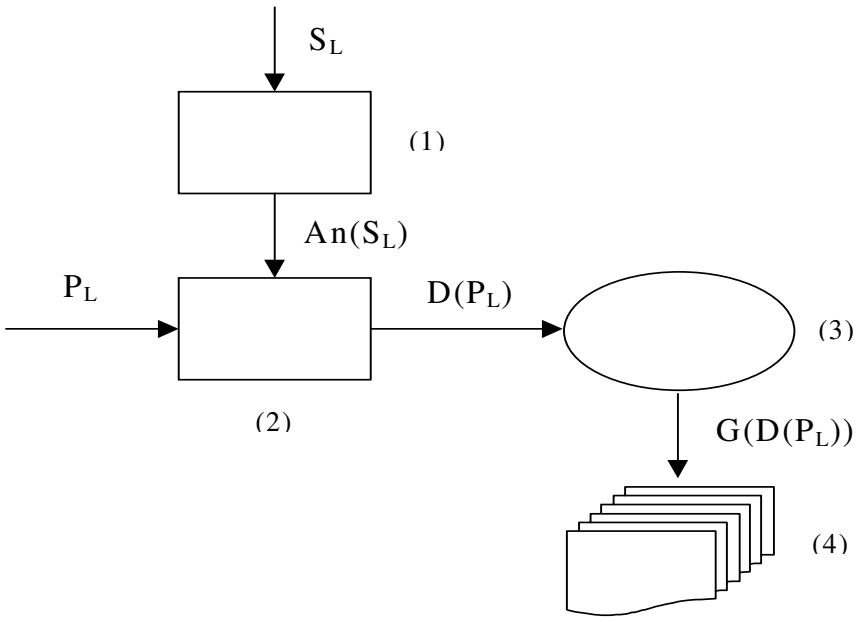


Figura 1 – Esboço geral da Arquitetura

onde temos como entrada para a arquitetura:

S_L: especificação, em semântica denotacional modificada para grafos[1], de uma linguagem qualquer L;

P_L: programa escrito na dada linguagem.

No esquema apresentado, (1) é o módulo que gera um analisador para detecção de dependências de um dado programa. Este módulo recebe como entrada a semântica de uma linguagem (S_L) e tem como saída um analisador ($An(S_L)$) para L, que detectará dependências para um programa em L. Resolvemos utilizar Semântica Denotacional pela naturalidade desta no projeto de linguagens [PAG81], o que facilita a construção de uma arquitetura genérica para a manipulação de construções destas linguagens.

[1]Semântica denotacional com saída para grafos de dependência (descrição na Seção Formato da Especificação Semântica)

O módulo (2) executa o analisador ($An(S_L)$) gerado no módulo (1), utilizando como entrada um programa (P_L) na linguagem do analisador. Este módulo tem como saída um arquivo ($D(P_L)$) com as dependências existentes no programa.

Em (3), fazemos a alocação do grafo de dependências ($G(D(P_L))$) a partir da saída do módulo (2). É de fundamental importância que utilizemos uma representação intermediária que seja adequada para a detecção de paralelismo.

Finalmente no módulo (4), temos a manipulação do grafo de dependências ($G(D(P_L))$) para a geração do código paralelo e recuperação na sintaxe da linguagem fonte.

Capítulo 2 – Conceitos Básicos

Neste capítulo, veremos os principais conceitos básicos necessários para o entendimento de todo o problema de detecção e geração automática de paralelismo.

2.1 - Paradigmas de Programação Paralela

2.1.1 - Troca de Mensagens

Neste paradigma, o usuário utiliza bibliotecas de funções que têm a incumbência de explicitamente compartilhar informações entre os processos através do envio e recebimento de mensagens. Este tipo de comunicação se torna necessário pois os processos têm acesso apenas a uma memória local. Esta abordagem é mais escalonável que *Memória Compartilhada,* mas tem como desvantagem um tempo maior no compartilhamento de informações.

2.1.2 - Paralelismo de Dados

Neste, o paralelismo é determinado pela distribuição dos dados. Ou seja, a natureza dos dados manipulados indicará se a operação poderá ser paralelizada ou não. Esta distribuição de dados é indicada através de construções de dados paralelas fornecidas pelo programador. O código com as construções é passado para um compilador que gera a comunicação necessária (transparente ao usuário) para a distribuição.

2.1.3 - Memória Compartilhada

A troca de informações é realizada neste paradigma através da utilização de um espaço de memória comum, que é compartilhado entre os processos, os quais precisam ser sincronizados para que não haja conflito. O acesso a esta memória comum pode ser

feito via um canal compartilhado ou via rede. Um dos problemas ocasionados por este tipo de conexão é a escalabilidade, uma vez que o acesso dos processadores ao canal deve ser sincronizado. Afim de diminuir os acessos à memória, computadores projetados desta maneira têm processadores que possuem uma memória cache.

2.1.4 - Threads

Um único processo tendo múltiplos caminhos de execução concorrentes. Por se tratar de um único processo, o compartilhamento de recursos (variáveis, arquivos, ..) é automático entre as *threads*, deixando para o programador a responsabilidade de sincronizar os acessos a estes recursos. A grande vantagem da utilização desta abordagem é manter a flexibilidade das soluções concorrentes sem ganhar o custo da complexidade do controle de processos [GRA98].

Além destas, *Pipes*, *Sockets*, RPC's também são comumente utilizadas. Estas abordagens, ou têm seus próprios mecanismos de sincronismos, ou utilizam outros, como semáforos. Uma descrição mais detalhada destas e de outras formas de programação e comunicação entre processos pode ser encontrada em [GRA98].

Todos estes modelos de programação são independentes da máquina/arquitetura. Ou seja, qualquer um pode ser implementado em qualquer *hardware,* dado que haja um sistema operacional com suporte apropriado. Uma implementação eficiente compatibiliza a máquina alvo e possibilita facilidades na programação [MHPCC]. Normalmente, uma abordagem de programação tende a espelhar a arquitetura de máquina mais apropriada [BAS96].

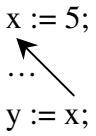
2.2 - Tipos de Dependências de Dados

Relações de dependências de dados são usadas pelos compiladores para representar as restrições de ordem essenciais entre comandos ou operações num programa [WOL96].

Veremos agora os 3 tipos básicos de dependências de dados.

2.2.1 - Dependência de Fluxo

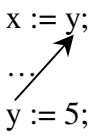
Ocorre quando uma variável é definida num comando e seu valor é utilizado num comando subseqüente.



Neste caso, a variável *y* recebe o valor da variável *x*, a qual foi definida num comando anterior.

2.2.2 - Anti-dependência

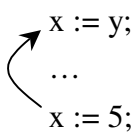
Surge quando o valor de uma variável é consultado num comando e, noutro subseqüente, esta é redefinida.



Aqui, a variável *y*, após ter seu valor consultado, é redefinida, forçando um sincronismo na execução das atribuições.

2.2.3 - Dependência de Saída

Uma dependência do tipo saída surge quando definimos uma variável num comando e a redefinimos num comando subseqüente.



Neste caso, *x* é definido num comando e redefinido num comando seguinte.

Podemos observar que tanto as anti-dependências quanto as dependências de saída surgem da reutilização (ou redefinição) de variáveis (espaço de memória). Estas são comumente chamadas de "falsas dependências". Nas dependências de fluxo por sua vez, isto não ocorre, pois o valor definido num primeiro comando é o mesmo valor consultado pelo segundo comando. Logo, este tipo de dependência é chamado de "dependência verdadeira".

Estes tipos de dependências também podem ser encontrados dentro de estruturas de repetição. Uma dependência nesta situação é *loop carried* se esta relaciona iterações diferentes de uma repetição, e é *loop independent* se a dependência ocorre numa mesma iteração.

2.3 - Representações Intermediárias

Usualmente, representações intermediárias são estruturas utilizadas pelos compiladores, as quais auxiliam processos de otimização, geração e paralelização de código, entre outros. Estas estruturas armazenam, normalmente, informações sintáticas, dependências de dados e de fluxo, existentes num programa. Em sua maioria, elas são a ligação entre o que o programador escreve e o que a máquina alvo entende.

Com o rápido crescimento de compiladores otimizadores e paralelizáveis, muitos pesquisadores esforçaram-se no projeto de representações intermediárias. As representações que serviram de base para estas pesquisas foram as seguintes.

2.3.1 - Grafos de Fluxo de Controle

Um grafo de fluxo de controle é um grafo direcionado com nós *start* e *end* distintos, tais que todos os nós são alcançáveis a partir do nó *start* e todos os nós têm um caminho até *end*. *Start* é o único nó sem antecessores e *end* o único sem sucessores. Nesta representação os nós, que são Blocos Básicos [ASU86], representam seqüências de comandos (sem desvios) como atribuições, e as arestas representam as possíveis transferências de controle entre os nós. Suponha o seguinte trecho de código, o qual será também usado como exemplo nas representações posteriores:

```
(S1) if p then
(S2)     for i := 1 to n do begin
(S3)         a := 10 + i;
(S4)         b := 2 * a;
(S5)     end do
(S6) else begin
(S7)     a := 4;
(S8)     b := 3 * a;
(S9) end if
```

Figura 2 – Código exemplo

O grafo de fluxo de controle correspondente a este trecho é o seguinte:

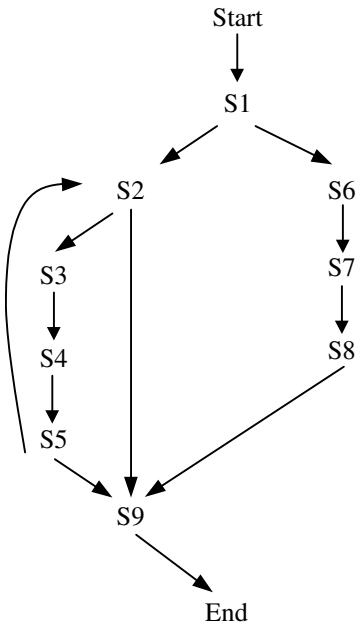
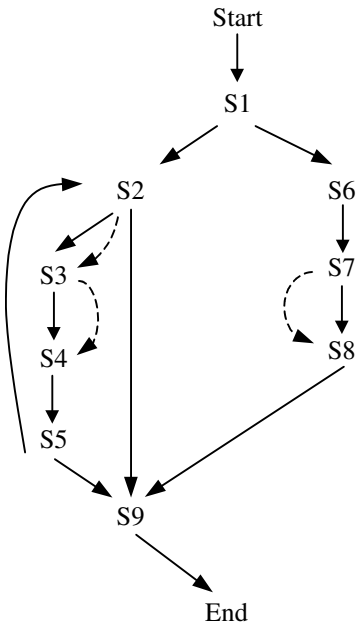


Figura 3 – Grafo de fluxo de controle da Figura 2

2.3.2 - Cadeias Definição-Uso

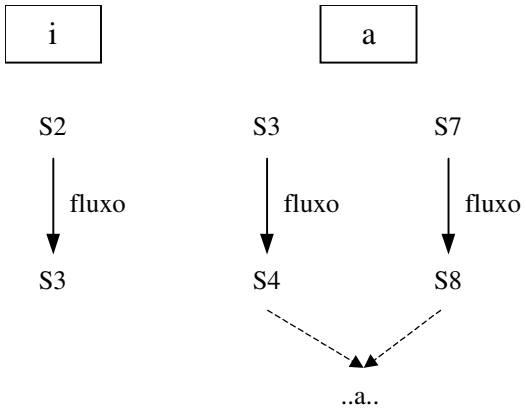
Cadeias Definição-Uso *(def-use chains)* são compostas por grafos que têm os mesmos nós que os grafos de fluxo. Entretanto, as arestas conectam cada definição de uma variável a todas as instruções que utilizam o valor definido. Como esta representação leva em consideração apenas dependências de valores (sem informação de dependência de controle), cadeias definição-uso são normalmente utilizadas em conjunto com algum outro tipo de representação. Por exemplo, grafos de fluxo de controle.



As arestas da cadeia definição-uso correspondentes à figura 3 estão representadas em tracejado, no grafo de fluxo de controle acima.

2.3.3 - Grafos de Dependências de Dados

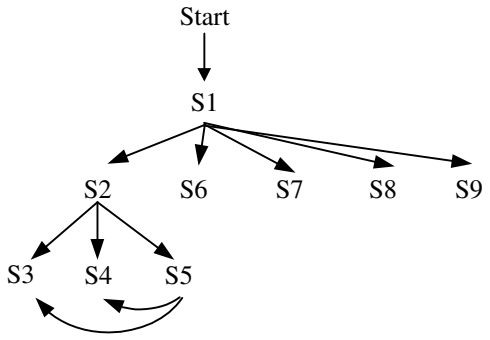
Estes tipos de grafos são uma generalização das cadeias definição-uso (dependência de fluxo). Nestes, encontramos não só arestas que indicam a relação definição-uso, como também uso-definição (anti-dependência) e definição-definição (dependência de saída). Estes tipos de grafos são utilizados por compiladores que realizam reorganizações completas de programas [KUC78]. O grafo de dependência de dados para o exemplo da Seção 2.3.1 é mostra a seguir.



Observe que os subgrafos associados às variáveis *p*, *n*, e *b* nem foram ilustrados, pois não existe nenhuma dependência no trecho de código dado. As arestas em tracejado acima indicam que, caso a variável *a* seja utilizada após os comandos S4 e S8, este uso terá ligação com as 2 (duas) definições da variável *a*.

2.3.4 – Grafos de Dependências de Controle

As relações de dependência de controle são uma maneira mais geral de se capturar condições essenciais de controle de execução de um programa [WOL96]. Por isso, nesta representação, caso a execução de um comando dependa direta ou indiretamente da execução de outro, estes nunca poderão executar em paralelo. Logo, grafos de dependência de controle determinam que comandos podem executar em paralelo devido à própria estrutura sintática do programa.



As arestas de S5 para S4 e S3 indicam a dependência de controle causa pelo comando de repetição (*for*).

2.4 – Especificação Semântica

Quando descrevemos uma linguagem de programação, precisamos nos preocupar não somente com a sintaxe (como aparenta as construções dessa linguagem), mas também com a semântica (significado das construções sintáticas). Uma descrição semântica, quando não é formal, leva a muitas ambigüidades, como por exemplo, ordem de execução de uma seqüência de chamadas de funções, tratamento de erros (situações inesperadas), … Estas ambigüidades são resolvidas somente na fase de implementação da linguagem, não obrigando os implementadores a uma uniformização na tomada destas decisões (resolução das ambigüidades). A solução atualmente adotada é a definição de uma semântica formal da linguagem, utilizando regras da lógica.

Várias abordagens têm sido utilizadas: Semântica Denotacional, Operacional, Axiomática, ... A fim de não fugirmos do escopo deste trabalho, falaremos apenas de Semântica Denotacional. Outras formalismos são encontrados em [PAG81].

2.4.1 – Semântica Denotacional

Mostraremos o formato geral de uma especificação formal em Semântica Denotacional através de um pequeno exemplo de expressões, as quais conterão apenas números e operações de soma e subtração.

Classicamente, uma descrição em Semântica Denotacional é dividida em 5 partes: Domínio Sintático

```
EPSILON:   Expr    ( expressoes )
OMICROM:   Oper    ( operadores )
PI:        Num     ( numeros )
```

, onde são declarados os não-terminais da gramática de entrada; Regras de Produção Abstratas

```
EPSILON   = EPSILON OMICROM EPSILON |
            '(' EPSILON ')' |
            PI;
PI        = number;
OMICROM   = '+' |
            '-';
```

, onde a gramática da linguagem é definida; Domínio Semântico

N = {…, -2, -1, 0, 1, 2, …}

, onde definimos os nomes dos domínios, bem como sua abrangência; Funções Semânticas

M: Expr → N → N
E: Expr → N → N

, que declaram os mapeamentos em relação aos domínios já declarados, e; Equações Semânticas

```
M [[ Expr ]] = E [[ Expr ]]
E [[ ( Expr1 ) ]] = E [[ Expr1 ]]
E [[ Expr1 + Expr2 ]] = E [[ Expr1 ]] + E [[ Expr2 ]]
E [[ Expr1 - Expr2 ]] = E [[ Expr1 ]] - E [[ Expr2 ]]
E [[ PI ]] = PI
```

, que dão significado às porções sintáticas da linguagem.

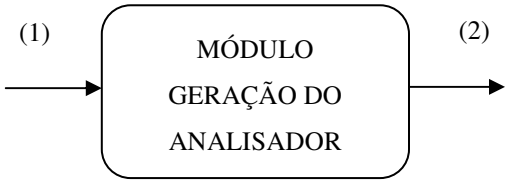
Uma diferença importante de ser entendida numa especificação é entre a linguagem tratada e a meta-linguagem (linguagem para definir linguagens) utilizada. Nas Equações Semânticas dadas acima, esta diferença é bem clara. No lado esquerdo das equações temos as construções sintáticas da linguagem e, do lado direito, temos os

relacionamentos entre estas construções. Ou seja, no exemplo acima, o sinal de soma (“+”) do lado esquerdo (referente à linguagem) é diferente do operador soma (“+”) utilizado do lado direito (referente à meta-linguagem).

Capítulo 3 – Descrição da Arquitetura

Como vimos na Figura 1, podemos dividir a arquitetura em 4 módulos distintos: Geração do Analisador, Detecção das Dependências, Geração do Grafo de Dependências e Detecção de Paralelismo, os quais serão descritos agora. Vale lembrar que o projeto desta arquitetura foi acompanhado pela construção de um protótipo. Ou seja, certas características da arquitetura são advindas de necessidades práticas.

3.1 - Geração do Analisador



Neste módulo, temos como entrada a descrição semântica para grafos da linguagem (1) e, como saída, temos um analisador para códigos na dada linguagem que, posteriormente, detectará as dependências de um programa nesta linguagem.

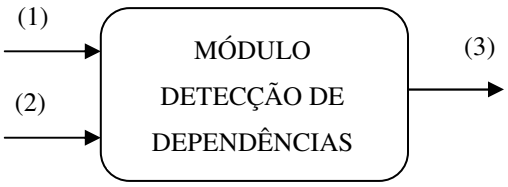
Para possibilitar a criação de uma arquitetura genérica, foi necessária uma padronização no formato da especificação da linguagem de entrada em Semântica Denotacional. O formato completo da especificação utilizada em nosso protótipo está descrito na Seção 3.7.1.

Além da genericidade estar relacionada às possíveis linguagens a serem processadas, também temos o termo "genérico" associado ao tipo de paralelismo (granularidade) obtido pela arquitetura. Como citamos anteriormente, utilizamos uma Semântica Denotacional modificada para grafos como nossa forma de especificação. Na verdade, esta faz o mapeamento entre as estruturas sintáticas e os grafos de dependências

(nossa representação intermediária). A parte da especificação fundamental para este mapeamento é a seção que descreve as Equações Semânticas, ou seja, a seção que dá significado às porções sintáticas da linguagem. Neste caso, a granularidade estará associada ao tipo de informação que colocamos no lado direito das Equações Semânticas de cada construção sintática. Por exemplo, as dependências indicadas para um comando de atribuição serão diferentes para granularidades do tipo de instrução e do tipo de bloco. Um exemplo geral é detalhado na Seção 3.7.

O analisador gerado constituir-se-á de um conjunto de funções que têm o papel de percorrer o código de entrada e gerar as dependências existentes neste. Como o processo de geração destas funções é automático, já que este é para qualquer especificação de uma linguagem, é preciso uma forma de obtenção destas funções. O formato para o nosso protótipo está descrito no Apêndice A.

3.2 - Detecção das Dependências



Agora veremos mais de perto, o funcionamento do módulo que detecta as dependências num dado código fonte.

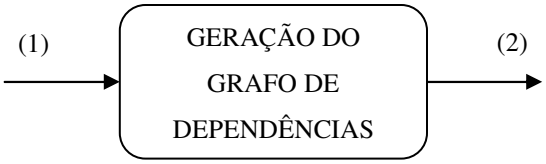
Este módulo recebe como entrada o analisador (1), que foi gerado pelo módulo anterior, o programa a ser analisado (2), e tem como saída as dependências encontradas no dado código (3).

Além destas entradas, funções que resolvem os lados-direito das regras também são passadas (ver Apêndice A). Estas não são geradas automaticamente (existem desde o início do processo). Apesar disso, esta não-automação não restringe a genericidade da

arquitetura. Isto apenas limita os operadores utilizados nos lados-direito (ver seção 3.7.1 para descrição completa dos operadores), os quais tem por finalidade fazer o mapeamento existente entre as estruturas sintáticas da linguagem e os nós no grafo de dependências. Como vimos, é neste ponto que a granularidade do código paralelo gerado é indicada.

A saída deste módulo é um arquivo texto que contém as dependências do programa fonte num formato previamente definido, baseado em expressões parentizadas. Neste formato, poderemos observar que não apenas informações de dependências são passadas para a construção do grafo. Dados como as regras sintáticas que geraram os comandos, trechos de código e identificadores de não-terminais, são também fornecidos e serão de fundamental importância para a recuperação do código na dada linguagem (a partir do grafo de dependências). Como falamos anteriormente, características como a divisão da arquitetura em módulos são advindas do protótipo desenvolvido. Logo, estas vão depender necessariamente do tipo de implementação utilizado. A descrição detalhada deste formato encontra-se no Apêndice B.

3.3 - Geração do Grafo de Dependências



Este módulo recebe como entrada um arquivo contendo as dependências encontradas no programa de entrada (1), e gera como saída o grafo correspondente alocado (2). Este também, além da geração do grafo propriamente dita, detecta ramos do código potencialmente paralelizáveis e também realiza otimizações, objetivando a eliminação de dependências que, consequentemente, possibilita maior facilidade no processo de paralelização.

Como destacamos anteriormente, esta arquitetura foi projetada em paralelo com o desenvolvimento de um protótipo. Ou seja, deverão haver características na arquitetura que são conseqüências pragmáticas. Logo, dependendo do tipo de ferramenta utilizada para instanciação da arquitetura, estas características podem ou não ocorrer. Um exemplo é a separação do módulo que detecta dependências com este que gera o grafo de dependências. No nosso protótipo, estes módulos foram postos em separado pois foram implementados em ferramentas diferentes.

Na escolha de uma representação intermediária, que neste caso funciona como uma ferramenta para auxílio na detecção e geração de trechos paralelizáveis, algumas questões precisam ser levadas em consideração:

⇨ A representação precisa conter informações tanto de dependência quanto de fluxo de dados existentes no programa;

⇨ Também precisa ser facilmente percorrida para busca de informações de dependência, o que irá facilitar na geração de código paralelo;

⇨ É interessante que esta representação seja flexível, a fim de possibilitar a geração de código eficiente para diversos elos plataforma-linguagem. Um dos pontos cruciais para a capacitação desta flexibilidade é quanto à granularidade do código paralelo tratado.

A escolha da representação intermediária de um programa tem um profundo impacto no projeto, na complexidade assintótica e na implementação de transformações de otimização e paralelização [PBJ91].

3.3.1 – Grafo de Dependências de Controle Descendente com Restrições

Veremos agora a representação intermediária chamada Grafo de Dependências de Controle "Descendente" com Restrições (*Constrained Forward Control Dependence Graph*) [WCH88]. Esta utiliza informações de dependências de controle e de dados para sua construção. Dependências de controle representam restrições sintáticas ao paralelismo, enquanto que restrições semânticas são expressadas através da dependências de dados.

Para exemplificar sua utilização, mostraremos um exemplo que é o método de Ordenação por Seleção. O código sequencial (em linguagem C) para este tipo de ordenação é mostrado abaixo:

Método de Seleção

```
(S1)   for (a=0; a<count-1; a++) {
(S2)     exchange = 0;
(S3)     c = a;
(S4)     t = item[a];
(S5)     for (b=a+1; b<count; b++) {
(S6)         if (item[b] < t) {
(S7)             c = b;
(S8)             t = item[b];
(S9)             exchange = 1;
                 }
           }
(S10)    if (exchange) {
(S11)        item[c] = item[a];
(S12)        item[a] = t;
           }
       }
```

O Grafo de Dependências de Controle Descendente para este código pode ser visualizado na Figura 3. Observe que apenas as informações de controle estão presentes no grafo. Fizemos isso com a intenção de facilitar o entendimento e destacar o possível paralelismo entre instruções já existente no nível sintático. Desta maneira, este grafo ainda não é chamado de Grafo com Restrições (semântica). A justificativa da palavra Descendente está no fato de não colocarmos no grafo as arestas que caracterizam uma repetição.

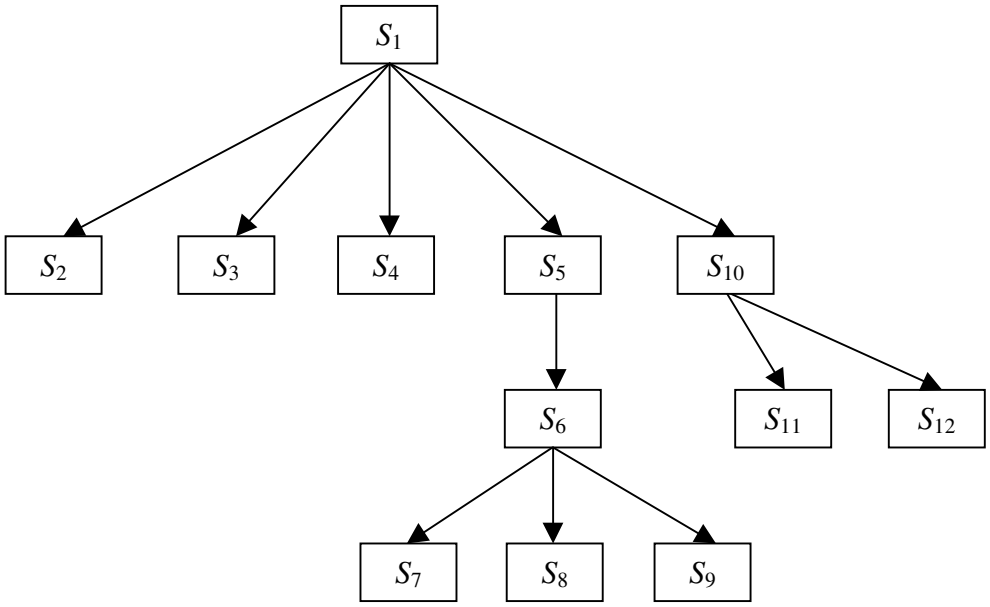


Figura 4 – Grafo de Dependência de Controle Descendente para o Método de Seleção

Este é o grafo de controle de fluxo "descendente" (*Control Forward Dependence Graph*) do algoritmo mostrado anteriormente. A construção deste tipo de grafo sugere que os nós pertencentes a um mesmo nível executem concorrentemente, satisfazendo, é claro, as dependências que existirem entre os nós e que não puderem ser resolvidas. Ou seja, a estrutura do grafo determina que porções sintáticas do código podem ser paralelizadas, e os valores semânticos (dependências de dados) determinam que porções podem realmente executar em paralelo. Obviamente, as dependências de dados, apesar de não estarem expressas no grafo, precisam ser respeitadas de forma que a semântica do

código paralelo obtido seja equivalente à do código sequencial original (Grafo de Dependências de Controle Descendente com Restrições). Um esboço do algoritmo que calcula as dependências de dados existentes entre as instruções é apresentado na seção 3.3.2.

Neste grafo, cada nó que contém filhos é um Ponto de Decisão onde, numa suposta execução, deverá ser feita a escolha entre as arestas de fluxo de controle existentes. Os tipos de nós que são Pontos de Decisão são condicionais (*if*, *if-else*, ..), iteradores (*for*, *while*, ..), e todos os comandos que ofereçam mais de um fluxo de execução.

Por se tratar de um grafo "descendente", não temos as arestas que indicam o retorno da execução no caso dos comandos de repetição. Com esta caraterística, os grafos de dependência de fluxo de controle "descendente", para programas estruturados, são normalmente árvores. Em sua maioria, as folhas destas árvores são comandos de atribuição. Isto pode variar de acordo com o tipo de granularidade indicado na semântica da linguagem de entrada.

A cardinalidade dos conjuntos de instruções a serem executadas paralelamente é uma indicação de qual ambiente melhor se adequa. No caso da granularidade fina, é interessante que adotemos memória compartilhada, na intenção de minimizar o custo na troca de mensagens. Situação oposta ocorre com paralelismo de granularidade mais alta, onde procuramos porções maiores de código que executem concorrentemente. Neste caso, dada uma baixa taxa de sincronização, devemos utilizar um ambiente distribuído.

Logo, é interessante que uma representação intermediária seja flexível o suficiente para se adaptar à plataforma, linguagem e granularidade de paralelismo abordadas.

3.3.2 – Algoritmo de Detecção de Dependências de Dados

Mostraremos agora um esboço do algoritmo que detecta as dependências de dados para grafos de dependência fluxo de controle, ou seja, restrições semânticas aplicadas à estrutura sintática do código.

Algoritmo de detecção de dependências:

Entrada: conjunto de instruções organizados de forma parentizada (refletindo a estrutura sintática do programa original) com variáveis de leitura/escrita já calculadas.

Saída: dependências calculadas entre as instruções.

Funções utilizadas: Testar dependências (*i*, *j*) – cria uma aresta entre os nós *i* e *j* caso haja dependência.

Propagar variáveis[2] – copiar as variáveis leitura/escrita que geraram dependências para o nó pai do nó *i.*

```
Para cada instrução
    Criar nó i                          // variáveis de leitura/escrita já calculadas
    Se nó i é Ponto-de-Decisão
        Chamar algoritmo (filhos de i) // descer na hierarquia de instruções
                                        // parentizadas (aumentar 1 nível no grafo)
    Fim-Se
    Para cada nó j (irmão de i) de 1 até (i – 1)  // irmão = mesmo nível
        Testar dependências anti, saída e fluxo ( i, j )   // baseado nas variáveis
                                                           // de leitura/escrita
    Fim-Para
    Propagar variáveis leitura/escrita
    i ← i + 1
  Fim-Para
Fim-Algoritmo
```

[2] Da figura 4, se tivermos dependências entre as instruções S_8 e S_{11}, estas dependências também precisarão existir quando testarmos se S_5 e S_{10}, que estão no mesmo nível (irmãos), podem ser concorrentes.

3.4 - Detecção de Paralelismo

Neste módulo, percorreremos o grafo de dependências alocado, que é a entrada deste módulo, identificando os ramos potencialmente paralelizáveis.

Na detecção do paralelismo, técnicas largamente conhecidas como Privatização (ver Seção 3.5), que visam eliminação de dependências do tipo anti e de saída, eliminação de definições (escritas) não alcançadas, entre outras, serão aplicadas na tentativa de aumentar o paralelismo presente nos programas.

Outro tipo de paralelismo detectado são as construções FORALL (comando *for* com iterações independentes). Um problema encontrado neste ponto são as expressões que indexam variáveis do tipo vetor. Suponha o seguinte trecho de código:

```
for ( i = 0; i < (MAX/2); i++ ) {
        VET [ i ] = i;
        VET [ i + (MAX/2) ] = 2*i;
}
```

Apesar de facilmente visualizarmos que as iterações deste comando de repetição são independentes, nosso protótipo não consegue detectá-lo. Isto ocorre porque nosso protótipo não avalia expressões da linguagem tratada, pois ela pode ser qualquer. Ou seja, não conseguimos saber por exemplo, para um dado contexto, se (i+j) é igual, maior ou menor que (j+k). Uma solução para este problema seria o mapeamento destas funções básicas (aritméticas, lógicas, …) para operadores na nossa meta-linguagem, mas acreditamos que a relação custo-benefício deste esquema não seria vantajosa.

Como comentado na seção anterior, a representação intermediária (grafo de dependências) e o nível de paralelismo tratado, são fatores fundamentais para a determinação do ambiente paralelo no qual o código será gerado. Por se tratar de uma arquitetura genérica, não podemos garantir a existência de um tipo de ambiente para

qualquer linguagem, de forma que esta implementação seja garantidamente eficiente. Por isso, devemos particularizar o código gerado para uma linguagem e plataforma específicas.

Na intenção de evitarmos momentaneamente a particularização do protótipo, criamos um primeiro passo onde geramos código paralelo alto-nível (independente da arquitetura alvo) e depois instanciamos para uma máquina específica. O código paralelo gerado será constituído de construções (comandos) paralelas de alto-nível, inicialmente. *Forall* é a construção que indica que as iterações de um comando de repetição são independentes, podendo ser executadas em paralelo. *Cobegin/coend* é um outro tipo de construção, onde blocos de instruções podem executar em paralelo.

Ou seja, neste módulo também fazemos a inserção das construções paralelas de alto-nível (*cobegin/coend, private, forall*) no código, afim de explicitarmos a independência (tanto de dados quanto de fluxo de controle) existente entre os comandos. Para isto, projetamos o algoritmo mostrado a seguir. Note que o código paralelo gerado pelo algoritmo descrito é intermediário, pois ainda não falamos da recuperação do código na sintaxe da linguagem original. Para viabilizar esta recuperação, as informações sintáticas passadas na construção do grafo de dependências serão repassadas para a saída deste algoritmo. Afim de termos um sistema homogêneo e facilitarmos o percorrimento, também demos o formato de expressões parentizadas para a saída deste algoritmo.

Algoritmo de geração do código paralelo:
Entrada: grafo com as dependências detectadas.
Saída: código intermediário paralelo correspondente *// cobegin/coend*
Imprime "cobegin"
Para cada nó *i* filho do nó corrente
Para cada nó *j* (irmão de *i*) de 1 até (*i* – 1)
Se ∃ dependência-fluxo nós(*i*, *j*)
Imprime "end"

```
                Imprime "coend"
                Imprime "cobegin"
                Imprime "begin"
                EliminaDependências nós ( qualquer-irmao de i, k )
                        // Como inserimos um ponto de sincronismo, podemos
                        //eliminar todas as dependências de um nó irmão qualquer
                        //com um nó anterior ao ponto de sincronismo
                Sair-Para j  // Sai do Para (for) mais interno (j)
            Else
                Se ∃ dependência-anti nós( i, j ) ou ∃ dependência-saída nós( i, j )
                    Acumula-Buffer ( variável nó ( i, j ) )  // Acumula num buffer as
                                                             //variáveis que geraram dependências
                                                             //tipo anti e saída
                Fim-Se
            Fim-Se
        Fim-Para      // j
        Se ~∃ dependencia-fluxo      // Se não existiu dependência de fluxo
            Se ∃ dependência-anti ou ∃ dependência-saída   // existiu !!!
                Imprime "end"
                Imprime "begin"
                Imprime "private" + Buffer-Acumulado  // variáveis privadas
            Fim-Se
        Fim-Se
        Caso ( tipo-nó ( i ) ):
            folha: GerarNóFolha ()
            ptoDecisão: GerarPtoDecisão ()      // Chama o alg. Recursivamente
                                                // Testar se é um Forall
        Fim-Caso
    Fim-Para      // i
Fim-Algoritmo
```

3.5 - Recuperação do Código

A principal dificuldade deste módulo é a recuperação do código (gerado pelo grafo) paralelo na sintaxe fiel da linguagem de entrada. Várias medidas precisam ser tomadas para a realização desta tarefa. Dentre elas, a passagem de informações sintáticas (identificadores de meta-variáveis, não-terminais, …) para o Grafo de Dependências e o percorrimento das regras de produção da linguagem.

Como a unidade mínima (não-paralelizável) de nossa arquitetura é o comando que determina a granularidade (ver seção 3.7.1), podemos ter como idéia a reconstrução da árvore sintática do programa a partir das "folhas" desta árvore (as unidades mínimas), as quais estarão codificadas juntamente com as construções paralelas *cobegin*/*coend*, *forall*, .., e serão a saída do algoritmo dado na seção anterior. Desta forma, precisamos passar informações ao Grafo para que possamos saber que regra gerou uma dada unidade mínima, que regra gerou a composição desta com outra unidade, e assim sucessivamente até a construção completa da árvore sintática.

O código seqüencial dado anteriormente (Seção 3.3.1) terá sua seguinte versão concorrente (após recuperarmos o código intermediário obtido do algoritmo da seção 3.4), utilizando-se granularidade a nível de comandos de atribuição:

Método de Seleção Concorrente

```
(S1)  for (a=0; a<count-1; a++)
      {
        cobegin
          begin
(S2)        exchange = 0;
          end
          begin
(S3)        c = a;
          end
          begin
```

```
(S4)            t = item[a];
            end
        coend
        cobegin
            begin
(S5)            for (b=a+1; b<count; b++)
                {
(S6)                if (item[b] < t)
                    {
                        cobegin
                            begin
(S7)                            c = b;
                            end
                            begin
(S8)                            t = item[b];
                            end
                            begin
(S9)                            exchange = 1;
                            end
                        coend
                    }
                }
(S10)           if (exchange)
                {
                    cobegin
                        begin
(S11)                       item[c] = item[a];
                        end
                        begin
                            private item[a] copyout
(S12)                           item[a] = t;
                        end
                    coend
                }
            }
        end
    coend
```

Obviamente, em situações, como no exemplo abaixo, onde no código gerado contiver uma seção cobegin/coend, e nesta termos apenas um único comando, este deverá ser executado seqüencialmente, evitando com isso o disperdício de tentarmos paralelizar uma verdadeira dependência (*).

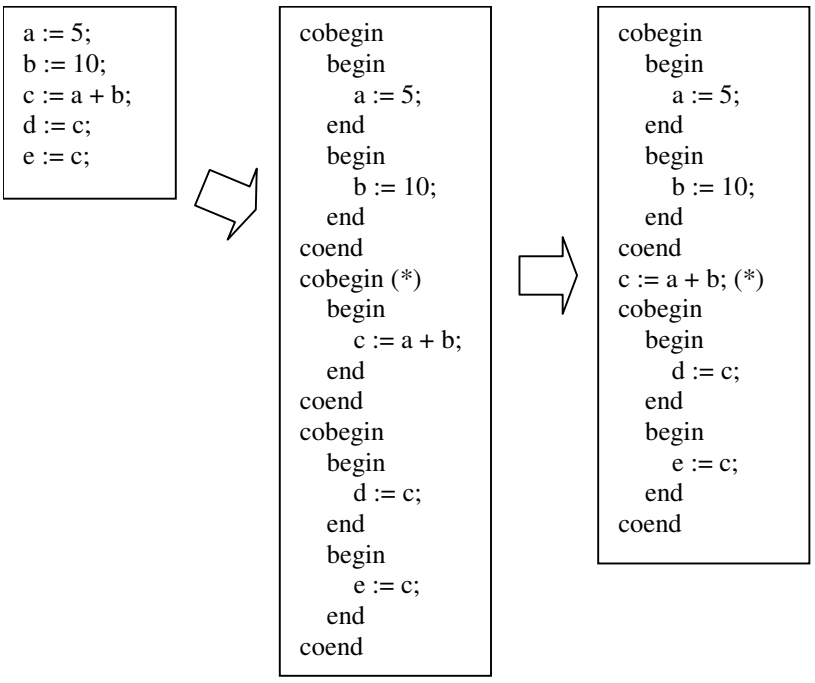


Figura 5 – Simplificação das seções cobegin/coend

Além das construções descritas acima, observamos também a presença da construção *private* aplicada ao *item[a]*;. Esta construção tem por objetivo aumentar os trechos de código paralelizáveis existentes num programa. A técnica chamada "Privatização" (*Privatization*) alcança esta otimização criando instâncias de variáveis específicas de processos. Este método tenta remover dependências do tipo anti e de saída, que se caracterizam pela reutilização de uma mesma locação de memória para uma variável. No exemplo, as instruções S_{11} e S_{12} podem executar paralelamente já que S_{12} define *item[a]* localmente, podendo usá-la ou alterá-la dentro de seu escopo, enquanto que S_{11} recebe uma cópia de *item[a]* para usar seu valor. Auxiliando o termo *private*, também temos o termo *copyout*, que indica o valor da variável tratada ao final do trecho *cobegin/coend*. Obviamente, esta definição só pode ser feita depois da execução de todos as instruções concorrentes da seção *cobegin/coend* respectiva.

Esta técnica também se adequa a qualquer granularidade. Basta apenas que identifiquemos quais são os valores "lidos" e/ou "escritos" para cada tipo de granularidade. Por exemplo, caso a granularidade seja a nível de procedimento (como no nosso exemplo – Seção 3.7.1), a seqüência de chamadas de procedimentos numa seção

cobegin/coend poderá sofrer o processo de "Privatização" de acordo com os parâmetros de entrada e saída existentes.

A simulação das construções de alto nível (*cobegin/coend*, *forall*, ..), para instruções reais na linguagem paralela destino, será realizada através de um pós-processamento. Logo, para este pós-processamento, também precisaremos de informações da plataforma destino. O grau de dificuldade desta simulação estará diretamente relacionado à afinidade da linguagem destino com estes tipos de instruções.

Um dos grandes problemas encontrados na implementação de um protótipo para esta arquitetura está relacionado à plataforma e linguagem que se deseja gerar o código paralelo. Não é comum encontrarmos bibliotecas para construção de código paralelo que utilizem o **mesmo** formato para várias linguagens. Esta **mesma** formatação é interessante já que trabalhamos com geração automática de código para uma arquitetura genérica. Com isto, somos obrigados a instanciar a arquitetura a um sistema em particular.

Comentamos que a adequação da representação intermediária ao elo linguagem-plataforma determina a eficiência do código gerado. Logo, precisamos particularizar a linguagem e, conseqüentemente, a plataforma de forma a obter um código interessante.

Na intenção de generalizarmos ao máximo o código final gerado (após o pós-processamento que fará a substituição das construções em alto-nível por comandos na linguagem paralela destino), podemos adotar um padrão seguido por parte dos construtores de ambientes paralelos. Um exemplo seria *PThread* [LEW96], que é um padrão POSIX[3] (*Portable Operating System Interface*) de API[4] (*Application Program Interface*) e que oferece uma biblioteca de rotinas usadas para o desenvolvimento de programas "multithreaded".

[3] Conjunto de comitês na IEEE que tem como objetivo a definição de API's que sejam comuns a todos os sistemas UNIX.

[4] Um conjunto de funções numa biblioteca, que contém suas assinaturas e semânticas.

3.6 – Pós-processamento

Nesta seção, faremos o mapeamento de nossas construções de alto-nível (FORALL, cobegin/coend, ..) para a linguagem paralela alvo[5]. Como citado na seção anterior decidimos, inicialmente, fazer o mapeamento para *threads*, que executará sobre um ambiente de memória compartilhada. Os argumentos favoráveis a isso são:

- Em termos do sistema de computação, teremos menor degradação do desempenho se compararmos criação e gerência de processos com *threads*;
- Compartilhamento automático de variáveis globais;
- *Threads* são a construção certa, do ponto de vista do sistema operacional, para suportar paralelismo de granularidade fina em multiprocessadores simétricos (SMP's) [MOR94].

Além destas características uma, que independeria da linguagem paralela utilizada, é não haver a necessidade de sincronizarmos os acessos às variáveis globais, já que, por construção, a arquitetura garante.

3.6.1 – Mapeamento das Construções Paralelas

3.6.1.1 - Forall

Nesta construção, criamos uma função (que tem como comandos o corpo do *for* original) que inicia a execução das *threads* e, no corpo original do *for*, disparamos a execução de diversas *threads*, sendo 1 (uma) para cada iteração do *for*.

[5] A linguagem seqüencial escolhida para o protótipo foi C, dada a facilidade de encontrar ambientes paralelos com *threads*, rodando sobre o ambiente UNIX.

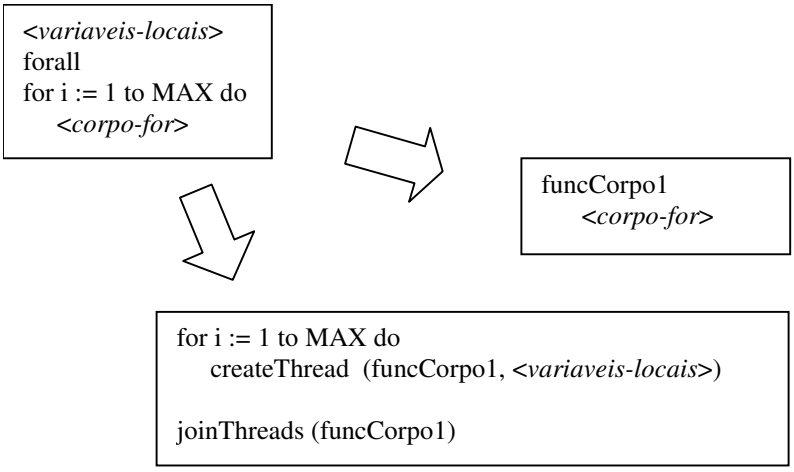


Figura 6 – Esquema de transformação do forall

Como estamos supondo um ambiente de memória compartilhada, não há a necessidade de passarmos as variáveis globais para a função que inicia a *thread*, e a passagem de variáveis locais dependerá da forma de tratamento de escopo da linguagem na ativação de procedimentos (estático ou dinâmico).

Só podemos continuar a execução (após o comando de repetição) depois que todas as iterações do *forall* tenham sido executadas, ou seja, após todas as *threads* terem terminado. Para isto, chamamos a função *joinThreads*, que funciona como um ponto de sincronismo.

3.6.1.2 – Cobegin/Coend

Para trechos *begin/end* de uma seção *cobegin/coend*, os quais podem executar em paralelo, temos como saída a geração de várias *threads*, 1 (uma) para cada trecho.

Note que, da mesma maneira que no comando de repetição, precisamos, após o disparo das *threads*, de um ponto de sincronismo, o qual também está indicado através do comando *joinThreads*.

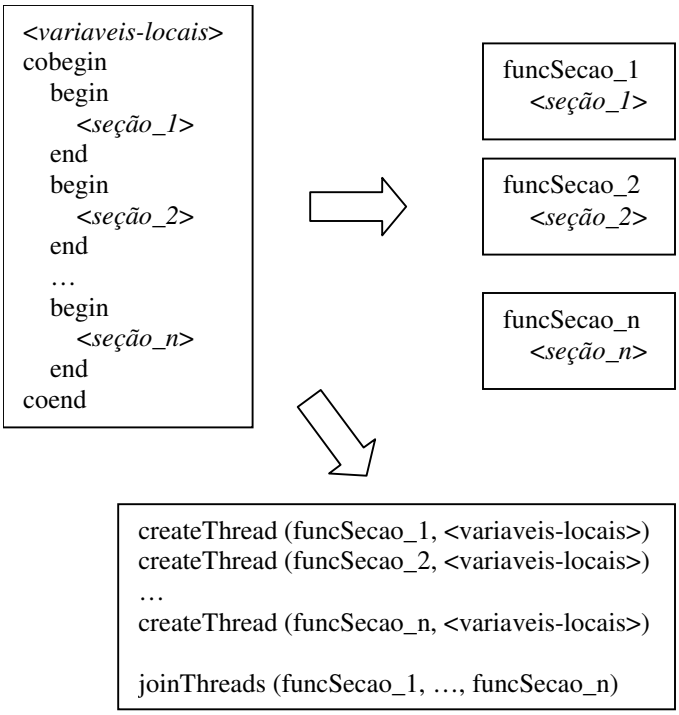


Figura 7 – Esquema de tradução das seções cobegin/coend

3.6.1.3 – Private

Como vimos anteriormente, esta construção tem como objetivo eliminar as dependências do tipo *anti* e de *saída*, que se caracterizam pela reutilização de uma locação de memória. Logo, seguindo o mesmo padrão de criação das *threads*, precisaremos sincronizá-las para que possamos ter os ramos com estes tipos de dependências executando em paralelo.

A solução adotada no protótipo é diferente da sugerida quando falávamos apenas em construções de alto nível. O esquema desta transformação pode ser visualizado abaixo:

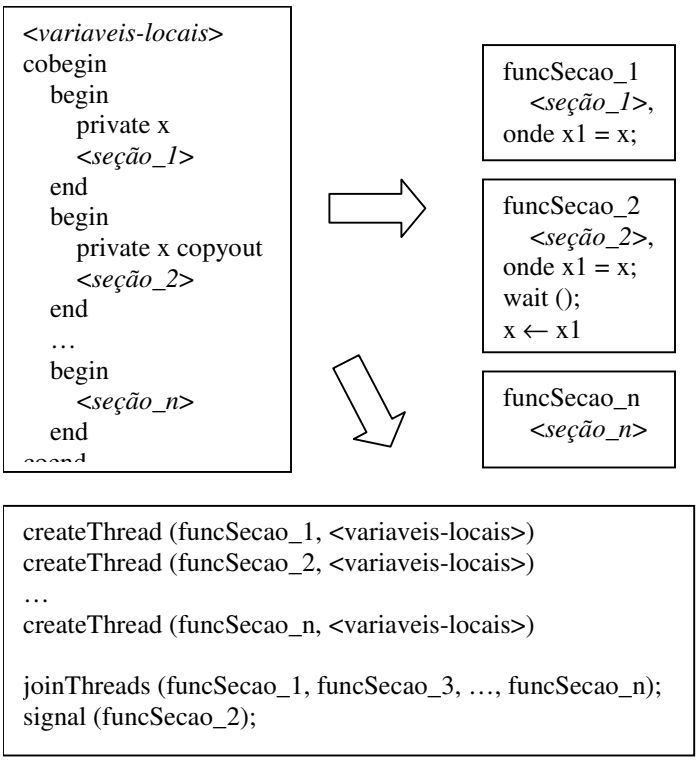


Figura 8 - Esquema de tradução das seções cobegin/coend com variáveis privadas

Neste esquema, estamos aplicando uma técnica chamada Renomeação [CF87b], que se caracteriza pela criação de pseudônimos para eliminação de falsas dependências (ver Seção 2.2). Neste, a variável *x* das seções seção_1 e seção_2 precisam ser privadas para que os módulos possam executar em paralelo. Utilizamos a palavra *copyout* para indicar qual é a última definição da variável, ou seja, qual será o valor de *x* ao final da seção *cobegin/coend*. Caso *x* seja uma variável global, esta será visível em toda parte do código, inclusive, dentro da função funcSecao_2 (onde temos a palavra *copyout*). Logo, não poderemos utilizar o próprio nome da variável (*x*), pois esta poderá estar sendo lida e/ou escrita em outros módulos.

Simbolizamos a sincronização através das funções signal() e wait(), indicando que a atribuição da variável local *x1* à variavel *x* só ocorrerá depois de todas as outras *threads* terem terminado seus processamentos.

Apesar de estarmos utilizando funções de sincronismo (signal() e wait()), garantimos que nunca ocorrerá *deadlock*. Podemos afirmar isto pois, no ponto de sincronismo (*joinThreads()*), esperamos apenas as *threads* que não têm variáveis privadas *copyout*. Após isto, sinalizamos as que ficaram bloqueadas. Neste ponto, também não precisamos nos preocupar com escritas numa mesma variável pois, ao final de uma seção *cobegin/coend*, a variável pode assumir apenas um valor, que é o último atribuído.

Obviamente, os identificadores de funções geradas, as quais auxiliarão na execução do programa em paralelo, precisam ser únicos. Logo, é preciso utilizar uma lei de formação que evite estes conflitos. Em nosso protótipo, a ferramenta utilizada - TXL - fornece tal função [TXL95].

3.7 - Exemplo Geral

Daremos agora uma pequena idéia de como o código deverá ser trabalhado nesta arquitetura (ver Figura 1). Exemplificaremos dois tipos de granularidade: comandos e procedimentos[6]. Para estes exemplos, utilizaremos uma linguagem que é subconjunto da linguagem PASCAL.

3.7.1 – Formato da Especificação Semântica

Para possibilitar a criação de uma arquitetura genérica, foi necessária uma padronização no formato da especificação da linguagem (entrada) em Semântica Denotacional. Com isto, tornou-se interessante a utilização de uma ferramenta que fosse adequada à manipulação de padrões. Em nosso protótipo, a ferramenta escolhida foi TXL [TXL95], que é uma linguagem que trabalha com transformação de código a partir do reconhecimento de padrões.

3.7.1.1 – Especificação Documentada

Veremos agora, uma breve descrição de como deverá ser a especificação semântica da linguagem de entrada. Esta especificação utiliza a mesma sintaxe abstrata definida em [GUE95], e contém informação suficiente para a detecção de dependências.

Classicamente, uma descrição em semântica denotacional é dividida em 5 partes: Domínio Sintático, onde são declarados os não-terminais da gramática de entrada; Regras de Produção Abstratas, onde a gramática é definida; Domínio Semântico, onde definimos os nomes dos domínios, bem como sua abrangência; Funções Semânticas, que declaram os mapeamentos em relação aos domínios já declarados, e; Equações Semânticas, que dão significado às porções sintáticas da linguagem. Estas denominações foram vistas na Seção 2.5.1.

Como precisamos gerar um analisador sintático para a linguagem, planejamos uma formatação para nossa meta-linguagem onde só utilizamos as Regras de Produção da linguagem (2o. parte). Posteriormente, precisaremos das Equações Semânticas, que determinarão as dependências existentes entre os comandos, ou seja, os mapeamentos entre comandos e nós no grafo de dependências. Resolvemos, como forma de aproveitar descrições já existentes e não omitir as construções que dão um caráter documentativo à Semântica Denotacional, manter todas as partes e utilizar somente as que realmente nos interessam. Utilizaremos a notação BNF para ilustrar cada parte da especificação, dando mais ênfase às regras gramaticais e às equações semânticas, que serão a base do nosso sistema.

Comentaremos cada seção da descrição semântica utilizando as expressões e o comando de atribuição do subconjunto de PASCAL utilizado.

---

[6] Na verdade, apesar de estarmos tratando esta especificação como a nível de chamadas de procedimento, esta especificação apenas não levará em consideração os comandos de atribuição.

- Domínio Sintático:

<DomSint> ::= { <ident> **":"**<ident> **"("** { <ident> } **")"** }

, onde o primeiro identificador é o nome do domínio sintático, seguido de sua meta-variável que, por sua vez, é seguida por uma lista de palavras que fazem um breve comentário sobre o domínio.

Exemplo: SIGMA: Commands (repetição de comandos)
SIGMA1: Command (comando de atribuição)
EPSILON: Expr (expressões)
OMICROM: Oper (operadores aritméticos)
NI: Id (identificadores)
PI: Num (números)

- Regras de Produção Abstrata:

<RegrasProd> ::= { <ident> **":"** [<ident> **"="**]
{ <terminais+nterminais> } [**"|"** | **";"**] }

, onde o primeiro identificador é a meta-variável associada à regra, que é seguida do nome da regra. A repetição de terminais e não-terminais descreve o lado direito da regra de produção.

Seguindo o exemplo tratado, temos:

STMT_LIST: SIGMA = SIGMA1 SIGMA |
STMT_LIST1: SIGMA1 ;

```
ASSIGN:        SIGMA1    = NI ':=' EPSILON ';' ;
EXPRESSION:    EPSILON   = EPSILON OMICROM EPSILON |
FATOR:                   '(' EPSILON ')' |
ID:                      NI |
NUM:                     PI ;
IDENT:                   NI =  id ;
NUMERO:                  PI  =  number ;
```

Observe que utilizamos tipos primitivos de TXL (id, number) também como tipos primitivos da linguagem de entrada. Esta foi a maneira que encontramos de representar os valores semânticos da linguagem na linguagem do transformador.

- Domínio Semântico:

<DomSem> ::= { <ident> **":"** <ident> **"="** { <terminais> } **";"** }

, onde o primeiro identificador é a meta-variável do domínio, o segundo é o nome do domínio e, a seqüência de terminais e não-terminais é a descrição do domínio. Como esta parte não é utilizada pelo programa, torna-se interessante apenas em termos de documentação.

Em nossa arquitetura, a semântica de um comando é a alteração no grafo de dependências que reflete a execução deste. Pensando-se desta maneira, o único Domínio Semântico que nos interessa é o Grafo de Dependências.

Exemplo:

Graph = { Representação[7] }

[7] Conjunto de símbolos que representam um grafo de dependências. Omitido por questões de claridade.

- Funções Semânticas:

<FuncSem> ::= { <ident> **“:”** <ident> **“➔“** { <terminais > } **“;”** }

, onde o primeiro identificador é a meta-variável da função, seguido do identificador da função e de uma seqüência de identificadores e operadores que fazem o mapeamento de um domínio sintático num domínio semântico.

Como citado no item anterior, o único Domínio Semântico que nos interessa é o Grafo de Dependências. Logo, os domínios das Funções Semânticas serão mapeamentos de grafos em grafos.

Utilizando o mesmo exemplo, temos:

S: Commands → Graph ➔ Graph;
S: Command → Graph ➔ Graph;
E: Expr → Graph ➔ Graph;

Observe que o mapeamento não obriga necessariamente que haja alteração. No exemplo citado, as expressões, que por si só não explicitam nenhuma dependência, não modificam o grafo. Logo, temos a função identidade neste caso. Não sendo necessariamente importante para o esquema, esta parte também torna-se útil na documentação do código.

- Equações Semânticas:

<EquSem> ::= { <ident> **"[["** <ident> **"]]"** [**"("** <ident> **")"** | <EquSem>]
"=" { < operadores+nterminais> } **";"** }

, onde o primeiro identificador indica a função semântica aplicada, o segundo indica sobre que estrutura sintática esta função é aplicada (nome da meta-variável), o terceiro é o ambiente corrente no qual a função está sendo ativada e, a repetição de operadores (ver tabela a seguir) e não-terminais é a ação semântica correspondente.

Como exemplo, temos:

S [[ASSIGN]] = NI0 ← E [[EPSILON0]] ;
E [[EXPRESSION]] = E [[EPSILON0]], E [[EPSILON1]] ;
E [[FATOR]] = E [[EPSILON0]] ;

Uma exigência, nada trabalhosa, é o fato das meta-variáveis do lado direito, como por exemplo, (EPSILON), precisarem ser indexadas (EPSILON0, EPSILON1). Esta é uma restrição apenas de implementação, já que a ordem de ocorrência dos não-terminais no lado sintático pode ser diferente da ordem de ocorrência no lado semântico. O formato dos nomes das meta-variáveis é o nome seguido da quantidade de ocorrências antes deste, nesta regra. Mesmo ocorrendo uma única vez, numa dada regra, esta precisa ser indexada. Por exemplo, se EPSILON ocorresse uma única vez, apareceria EPSILON0 no lado direito.

Outro fato a ser observado, no exemplo citado, é a ausência do ambiente. Isto significa que o comando será executado diretamente sobre o Grafo de Dependências, e não sobre o resultado da execução de outro comando, como o exemplo abaixo:

S [[STMT_LIST]] = S [[SIGMA0]] (S [[SIGMA10]]) ;

, onde SIGMA0 é uma lista de comandos a ser executada sobre o ambiente gerado pela execução do comando SIGMA10, que por sua vez será executado sobre o Grafo, já que o ambiente não foi explicitado.

Observe que, para especificação das equações semânticas, utilizamos operadores da nossa meta-linguagem (←, (), etc). A descrição dos operadores disponíveis encontra-se na tabela abaixo:

id [[id]]	Chamada de uma equação. O primeiro identificador é a função semântica a ser ativada e o segundo é a meta-variável a ser passada (estrutura sintática).
<-	Operador que assemelha-se à uma atribuição, que cria uma dependência entre o lado esquerdo e o lado direito da seta.
-> ,	Operador associado ao condicional, que recebe uma expressão booleana do lado esquerdo, e comandos do lado direito. Os comandos associados ao ramos "if-else" são separados por uma vírgula.
[[ ]], …, [[ ]]	Utilizados para se retornar uma lista de dependências, cada uma associada à uma chamada de função. Uma situação interessante para seu uso é o caso dos operadores (binários, ternários, ..) usados nas expressões. Em nosso caso, o que se deseja é uma lista de dependências associadas à expressão.
[[ ]] ([[ ]])	Na composição de funções, o que temos é a execução da função que está parentizada, para posterior execução da função externa, sobre o ambiente resultante (*eager evaluation*).
<->	Este operador faz associação do corpo de um procedimento ao seu identificador e parâmetros. A informação existente neste corpo estará associada ao tipo de dependência buscada.
< >	Determinam sobre que ambiente um determinado conjunto de instruções será executado. O nosso conceito de ambiente contém apenas as declarações, já que, numa chamada de procedimento, precisamos saber

	qual é o grafo associado a este.
nil	Esta palavra é utilizada quando a estrutura sintática analisada, por si só, não tem significado relevante em termos de dependência (Exs.:'+', '-', '>', ..,).
forward	Caso a estrutura sintática tenha relevância apenas para as regras sintáticas "superiores" (regras que derivaram na regra corrente), a palavra reservada *forward* é utilizada, indicando que os valores semânticos serão tratados por regras superiores.
>>> <<<	Indica que tipo de nó será insignificante para a granularidade de um protótipo. Estes marcadores devem envolver os lados-direito das Equações Semânticas.

3.7.1.2 – Especificações completas do subconjunto de PASCAL

Um exemplo de especificação semântica para a linguagem (subconjunto de PASCAL) que contém expressões, comandos de atribuição, procedimentos, condicional, tem a seguinte aparência:

```
PSI: Prog        ( programas  )
DELTA: Decl      ( declaracoes )
IOTA: List       ( listaidents )
EPSILON: Expr    ( expressoes )
SIGMA: Stmt      ( comandos )
NI: Id           ( identificadores )
PI: Num          ( numeros )
ALFA: OpSoma     ( soma )

@@

PROGRAM:   PSI          = 'program' NI ';' 'var' DELTA SIGMA ;
```

```
DECS1:          DELTA     = DELTA1 ';' DELTA |
DECS2:                    DELTA1 ';' ;
VARS1:          DELTA1    = IOTA ':' NI |
VARS2:                    'proc' NI '(' DELTA ')' SIGMA ;
LISTIDS1:       IOTA      = NI ',' IOTA |
LISTIDS2:                 NI ;
STMT_LIST1:     SIGMA     = SIGMA1 SIGMA |
STMT_LIST2:               SIGMA1 ;
ASSIGN:         SIGMA1    = NI ':=' EPSILON ';' |
IF:                       'if' KAPPA 'then' SIGMA |
CALL:                     'call' NI '(' IOTA ')' ';' |
FOR:                      'for' NI ':=' EPSILON 'to' EPSILON 'do' SIGMA ';' ;
COMP:           KAPPA     = EPSILON RO EPSILON ;
EXPRESSION:     EPSILON   = EPSILON OMICROM EPSILON |
FATOR:                    '(' EPSILON ')' |
ID:                       NI |
NUMERO:                   PI ;
IGUAL:          RO        = '=' |
DIFERENTE:                '<>' |
MENOR:                    '<' |
MENOR_OU_IGUAL:           '<=' |
MAIOR:                    '>' |
MAIOR_OU_IGUAL:           '>=' ;
MAIS:           OMICROM   = '+' |
MENOS:                    '-' |
VEZES:                    '*' |
DIVIDE:                   '/' ;
IDENT:          NI        = id |
IDENT1:                   id '[' EPSILON ']' ;
NUM:            PI        = number ;
```

@@

lambda: Graph = Repres[8] ;

@@

M: Prog → Graph ➔ Graph;

D: Decl → Graph ➔ Graph;

I: List → Graph ➔ Graph;

S: Stmt → Graph ➔ Graph;

E: Expr → Graph ➔ Graph;

C: Comp → Graph ➔ Graph;

O: Opr → Graph ➔ Graph;

R: Rel → Graph ➔ Graph;

@@

M [[PROGRAM]] = NI0 < D [[DELTA0]] > <-> S [[SIGMA0]];

S [[STMT_LIST1]] = S [[SIGMA0]] (S [[SIGMA10]]) ;

S [[STMT_LIST2]] = S [[SIGMA1]] ;

D [[DECS1]] = D [[DELTA0]] (D [[DELTA10]]) ;

D [[DECS2]] = D [[DELTA10]] ;

D [[VARS1]] = I [[IOTA0]] ;

D [[VARS2]] = NI0 < D [[DELTA0]] > <-> S [[SIGMA0]] ;

I [[LISTIDS1]] = NI0 , I [[IOTA0]] ;

I [[LISTIDS2]] = NI0 ;

S [[ASSIGN]] = NI0 <- E [[EPSILON0]] ;

S [[IF]] = E [[KAPPA0]] -> S [[SIGMA0]] ;

S [[CALL]] = call NI0 < I [[IOTA0]] > ;

S [[FOR]] = NI0 E [[EPSILON0]] E [[EPSILON1]] -> S [[SIGMA0]] ;

[8] Seqüência de caracteres que representam um grafo de dependências

E [[EXPRESSION]] = E [[EPSILON0]], E [[EPSILON1]] ;

E [[FATOR]] = E [[EPSILON0]] ;

E [[ID]] = forward;

E [[NUMERO]] = forward ;

E [[IDENT]] = forward ;

E [[NUM]] = forward ;

R [[IGUAL]] = nil ;

…

R [[MAIOR_OU_IGUAL]] = nil ;

O [[MAIS]] = nil ;

…

O [[DIVIDE]] = nil ;

Observe que, nas Equações Semânticas, quando a estrutura sintática analisada não tem significado relevante (de acordo com a granularidade desejada) em termos de dependência (‘+’, ‘-‘, ‘>’, .., neste exemplo), colocamos a palavra *nil* como resultado da aplicação da função semântica. Caso a estrutura sintática tenha relevância apenas para as regras sintáticas “superiores” (regras que derivaram na regra corrente), a palavra reservada *forward* é utilizada, indicando que os valores semânticos serão tratados por regras superiores. Os caracteres “@@” são apenas limitadores das seções que auxiliam o *parser* do protótipo.

A granularidade do código paralelo a ser gerado está indicada nesta Especificação Semântica. No exemplo da linguagem dada, a granularidade indicada na especificação é a nível de comandos de atribuição (menor unidade para a nossa arquitetura), visto que nenhum comando utilizou os caracteres marcadores (>>> <<<). Caso queiramos que a granularidade do código passe a não levar em consideração as atribuições e, por conseqüência, também “ignore” os comandos compostos que contenham apenas atribuições, a especificação será a seguinte:

```
PSI: Prog          ( programas  )
DELTA: Decl        ( declaracoes )
IOTA: List         ( listaidents )
EPSILON: Expr      ( expressoes )
SIGMA: Stmt        ( comandos )
NI: Id             ( identificadores )
PI: Num            ( numeros )
ALFA: OpSoma       ( soma )

@@

PROGRAM:     PSI       = 'program' NI ';' 'var' DELTA SIGMA ;
DECS1:       DELTA     = DELTA1 ';' DELTA |
DECS2:                 DELTA1 ';' ;
VARS1:       DELTA1    = IOTA ':' NI |
VARS2:                 'proc' NI '(' DELTA ')' SIGMA ;
LISTIDS1:    IOTA      = NI ',' IOTA |
LISTIDS2:              NI ;
STMT_LIST1:  SIGMA     = SIGMA1 SIGMA |
STMT_LIST2:            SIGMA1 ;
ASSIGN:      SIGMA1    = NI ':=' EPSILON ';' |
IF:                    'if' KAPPA 'then' SIGMA |
CALL:                  'call' NI '(' IOTA ')' ';' |
FOR:                   'for' NI ':=' EPSILON 'to' EPSILON 'do' SIGMA ';' ;
COMP:        KAPPA     = EPSILON RO EPSILON ;
EXPRESSION:  EPSILON   = EPSILON OMICROM EPSILON |
FATOR:                 '(' EPSILON ')' |
ID:                    NI |
NUMERO:                PI ;
IGUAL:       RO        = '=' |
```

```
DIFERENTE:                '<>' |
MENOR:                    '<' |
MENOR_OU_IGUAL:           '<=' |
MAIOR:                    '>' |
MAIOR_OU_IGUAL:           '>=' ;
MAIS:       OMICROM       = '+' |
MENOS:                    '-' |
VEZES:                    '*' |
DIVIDE:                   '/' ;
IDENT:      NI            = id |
IDENT1:                   id '[' EPSILON ']' ;
NUM:        PI            = number ;

@@

lambda: Graph = Repres[9] ;

@@

M: Prog → Graph ➔ Graph;
D: Decl → Graph ➔ Graph;
I: List → Graph ➔ Graph;
S: Stmt → Graph ➔ Graph;
E: Expr → Graph ➔ Graph;
C: Comp → Graph ➔ Graph;
O: Opr  → Graph ➔ Graph;
R: Rel  → Graph ➔ Graph;

@@
```

[9] Seqüência de caracteres que simbolizam um grafo de dependências

```
M [[ PROGRAM ]] = NI0 < D [[ DELTA0 ]] > <-> S [[ SIGMA0 ]];
S [[ STMT_LIST1 ]] = S [[ SIGMA0 ]] (S [[ SIGMA10 ]]) ;
S [[ STMT_LIST2 ]] = S [[ SIGMA1 ]] ;
D [[ DECS1 ]] = D [[ DELTA0 ]] (D [[ DELTA10 ]]) ;
D [[ DECS2 ]] = D [[ DELTA10 ]] ;
D [[ VARS1 ]] = I [[ IOTA0 ]] ;
D [[ VARS2 ]] = NI0 < D [[ DELTA0 ]] > <-> S [[ SIGMA0 ]] ;
I [[ LISTIDS1 ]] = NI0 , I [[ IOTA0 ]] ;
I [[ LISTIDS2 ]] = NI0 ;
S [[ ASSIGN ]] = >>> NI0 <- E [[ EPSILON0 ]] <<< ;
S [[ IF ]] = E [[ KAPPA0 ]] -> S [[ SIGMA0 ]] ;
S [[ CALL ]] = call NI0 < I [[ IOTA0 ]] > ;
S [[ FOR ]] = NI0 E [[ EPSILON0 ]] E [[ EPSILON1 ]] -> S [[ SIGMA0 ]] ;
E [[ EXPRESSION ]] = E [[ EPSILON0 ]], E [[ EPSILON1 ]] ;
E [[ FATOR ]] = E [[ EPSILON0 ]] ;
E [[ ID ]] = forward;
E [[ NUMERO ]] = forward ;
E [[ IDENT ]] = forward ;
E [[ NUM ]] = forward ;
R [[ IGUAL ]] = nil ;
…
R [[ MAIOR_OU_IGUAL ]] = nil ;
O [[ MAIS ]] = nil ;
…
O [[ DIVIDE ]] = nil ;
```

Observe que as alterações que determinam a granularidade do paralelismo se restringem às Equações Semânticas. O tratamento da meta-variável do tipo ASSIGN

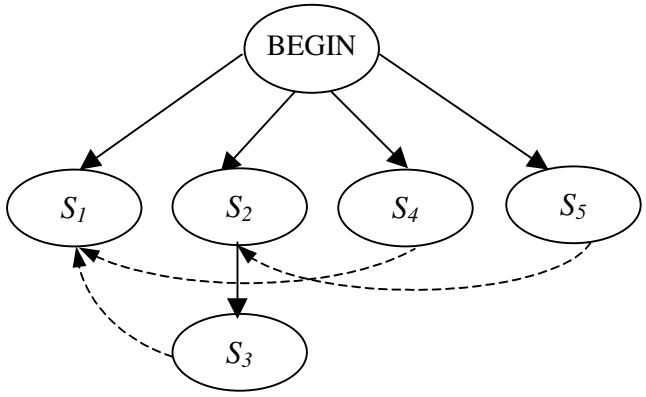
(inserção de informações de dependência no grafo) passa a não ter importância (vista isoladamente), já que a granularidade de paralelismo desejada não considera atribuição.

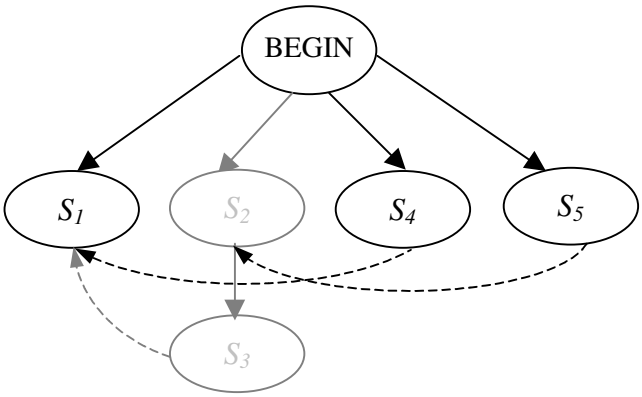
3.7.2 – Geração e Manipulação do Grafo de Dependências

Para exemplificarmos a geração e manipulação dos grafos de dependências, utilizaremos o código mostrado abaixo (baseado na linguagem especificada na seção anterior), que apenas declara 2 vetores e faz manipulações entre estes.

```
program P;
var i : integer;
   V1[10], V2[10] : integer;
proc INIT (val : integer, A[10] : integer)
    for i := 1 to 10 do
        A[i] := val; ; ;
proc MANIPULAR (A[10] : integer)
    for i := 1 to 10 do
        … A[i] …; ; ;
(S1)  call INIT ( 0, V1 );
(S2)  for i := 1 to 10 do
(S3)      V2[i] := V1[11 - i] ; ;
(S4)  call MANIPULAR ( V1 );
(S5)  call MANIPULAR ( V2 );
```

Abaixo encontramos os dois grafos gerados associados aos tipos de granularidade, na representação vista anteriormente (Seção 3.3.1), e os códigos gerados (em alto nível) respectivamente.

Granul.	Representação Gerada	Código Paralelo Obtido
INSTRUÇÃO		call INIT (O, V1); cobegin begin *forall* i:=1 to 10 do V2 [i] := V1 [11-i]; ; end begin call MANIPULAR (V1); end coend call MANIPULAR (V2);

PROCEDIMENTO		call INIT (O, V1);
		for i:=1 to 10 do
		V2 [i] := V1 [11-i];
		;
		cobegin
		begin
		call MANIPULAR (V1);
		end
		begin
		call MANIPULAR (V2);
		end
		coend

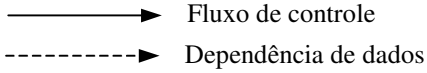


Figura 9 – Representação intermediária gerada e o código paralelo correspondente[10]

Informações de dependências, tanto de fluxo quanto de controle, são fornecidas pelo analisador (gerado a partir da especificação dada na seção 3.7.1), que tem sua forma de funcionamento descrita no Apêndice A. Estes dados são passados para este módulo utilizando-se o formato descrito no Apêndice B. Como comentamos anteriormente, certas características são naturais apenas deste protótipo desenvolvido, as quais são

conseqüências das ferramentas utilizadas. Nada impede que implementemos outro protótipo onde, por exemplo, a obtenção das informações de dependências e a manipulação do grafo sejam feitas em um único passo. Nesta situação, não precisaríamos desta representação textual dos grafos.

O trecho do grafo sombreado acima indica que **não** será criado um novo processo para o ramo. Apesar disso, este código precisará continuar sendo gerado, e as dependências respeitadas, para que a semântica do código paralelo seja equivalente à do seqüencial.

Observe que não colocamos os códigos referentes aos procedimentos os quais, de acordo com a granularidade desejada, também precisam ser modificados. Para este exemplo, o corpo dos procedimentos seria alterado apenas para a granularidade a nível de comandos de atribuição, que passaria a ter uma repetição do tipo *forall*.

3.7.3 – Código pós-processado

O código final (após o pós-processamento) terá o seguinte o formato, baseando-se nas regras de mapeamento apresentadas anteriormente (seção 3.6.1):

[10] O código paralelo obtido já está devidamente enxuto, como mostrado na Figura 4

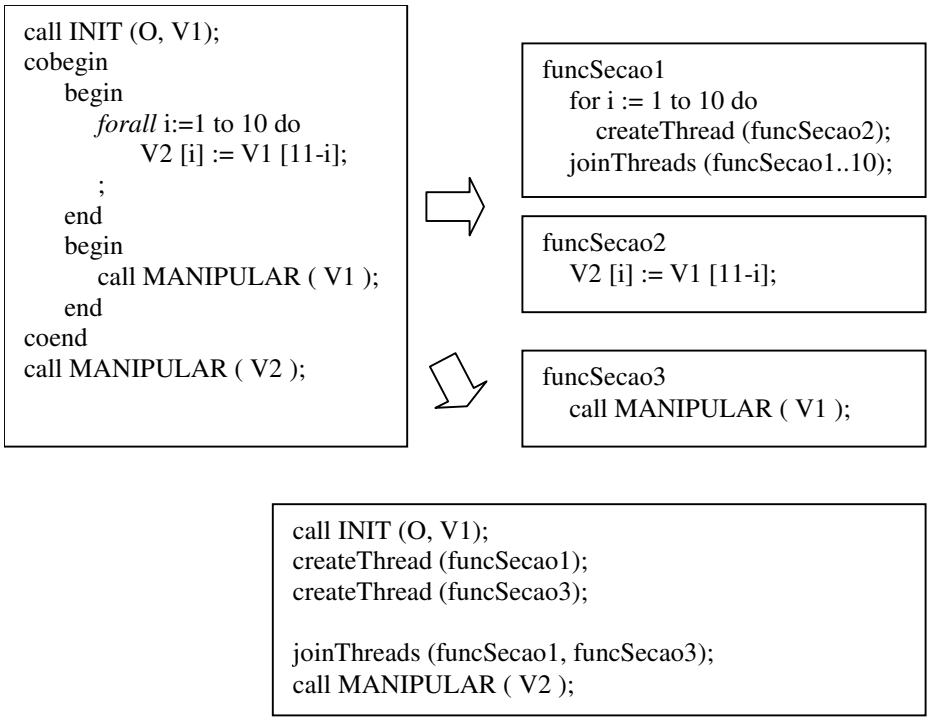


Figura 10 – Tradução do código paralelo baseado na especificação semântica com granularidade de instrução

Este é o código para o exemplo em que tratávamos da granularidade a nível de instrução. Novamente, não colocamos o código referente aos procedimentos (INIT e MANIPULAR) a fim de facilitar o entendimento. Como podemos ter aninhamento de seções paralelas (cobegin/coend), precisamos também tratar as funções geradas. Neste exemplo, a função *funcSecao1* tinha uma repetição do tipo *forall*. Como a granularidade tratada é a nível de instrução, e temos uma instrução dentro da repetição, expandimos também o nosso *forall,* gerando a função *funcSecao2*.

O exemplo referente a granularidade a nível de chamadas de procedimento pode ser visualizado abaixo.

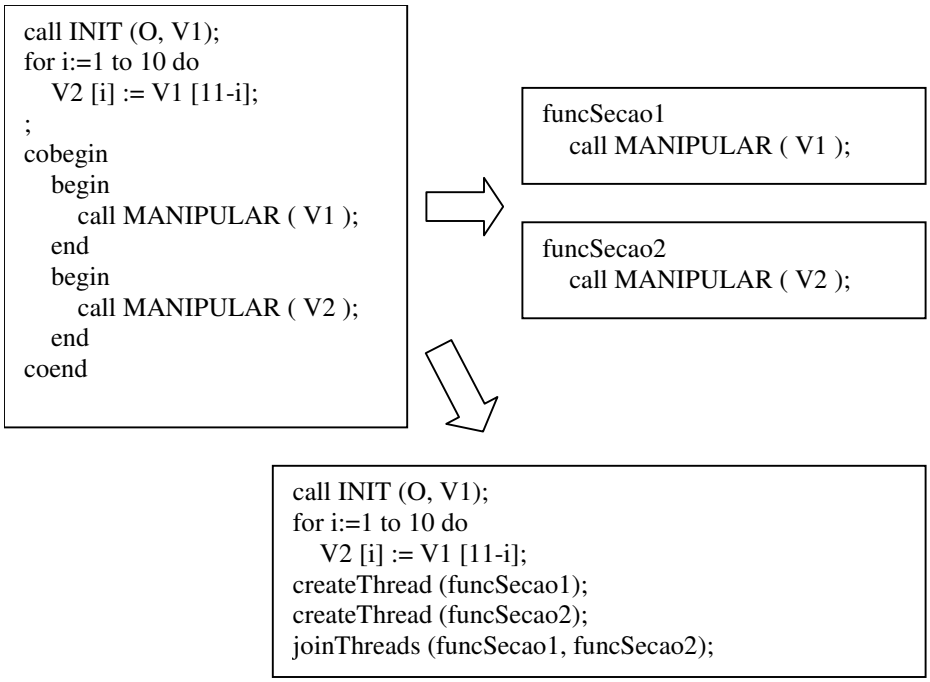


Figura 11 – Tradução do código paralelo baseado na especificação semântica com granularidade de chamadas de procedimentos

Observe que, como o comando de repetição acima não contém comandos que compõem a granularidade tratada (chamadas de procedimentos), este não foi transformado num *forall.*

Capítulo 4 – Testes Computacionais

Neste capítulo mostraremos todas as vantagens e desvantagens da instanciação do protótipo (em linguagem C) para arquiteturas *multithreaded*. Para tal, descreveremos 2 problemas abordados e como o protótipo conseguiu paralelizá-lo. A especificação semântica do subconjunto da linguagem C tratado é encontrada no Apêndice C. Note que o código gerado ainda pode ser bastante otimizado. Apresentamos o código desta maneira (sem otimizações) a fim de facilitar o entendimento.

4.1 – Multiplicação de matrizes

No problema de multiplicação de matrizes, o que temos é a geração de uma matriz C resultante do produto escalar das componentes das matrizes A e B da seguinte forma:

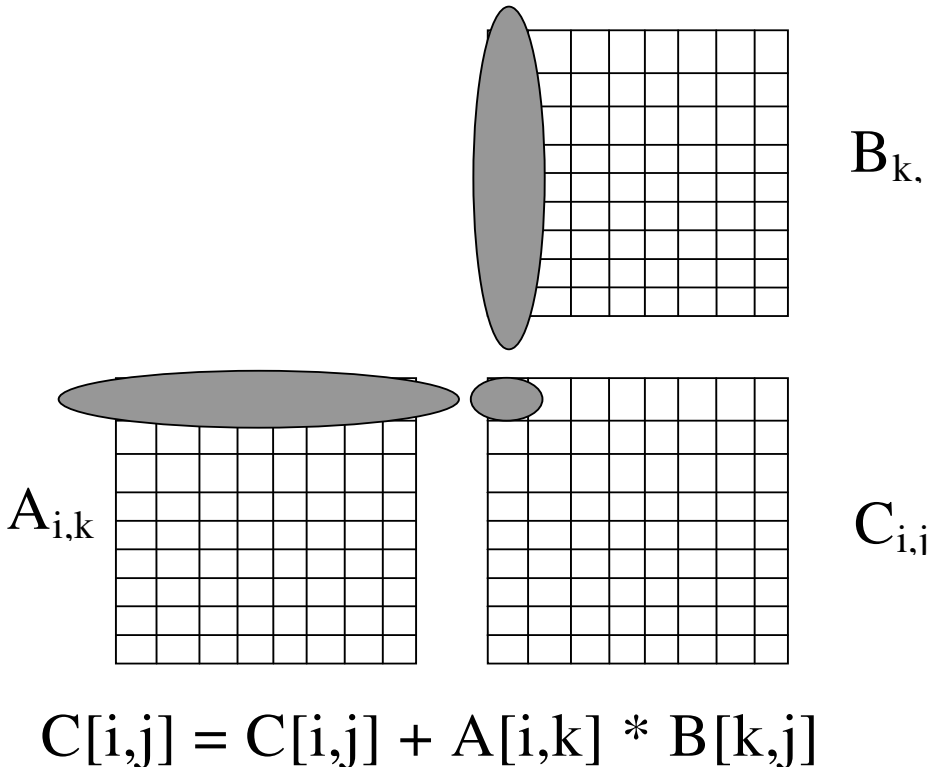


Figura 12 - Esquema de multiplicação de matrizes

Um código exemplo para este problema é o seguinte:

```
void main () {
    int i, j, k, tam;
    tam = 20;
    for (i=0; i<tam; i=i+1) {
        for (j=0; j<tam; j=j+1) {
            for (k=0; k<tam; k=k+1) {
               C[i][j] = C[i][j] + (A[i][k] * B[k][j]);
            } } }
}
```

O grafo de dependências para este código (obtido pelo algoritmo da seção 3.3.2) pode ser visualizado abaixo:

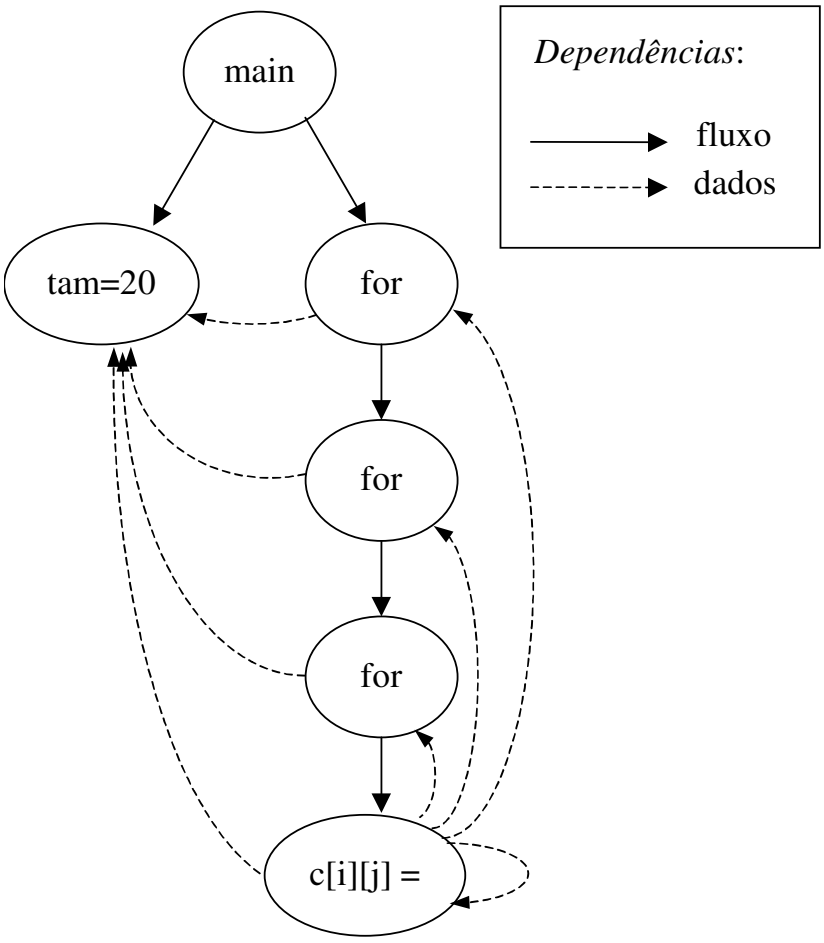


Figura 13 - Grafo de dependências de fluxo para multiplicação de matrizes

O código intermediário, que foi obtido através da aplicação do algoritmo da seção 3.4, é o seguinte:

```
void main () {
    int i, j, k, tam;
    cobegin
        begin
            tam = 20;
        end
    coend
    cobegin
        begin
            forall
            for (i=0; i<tam; i=i+1)
            {
                for (j=0; j<tam; j=j+1)
                {
                    for (k=0; k<tam; k=k+1)
                    {
                        C[i][j] = C[i][j] + (A[i][k] * B[k][j]);
                    } } }
        end
    coend
}
```

Podemos observar do código resultante que o sistema conseguiu detectar que o comando *for* mais externo (*i*) tem suas iterações independentes, ou seja, a dependência de dados gerada apartir da atribuição *C[i][j] = C[i][j] + (A[i][k] * B[k][j])* é particular a cada iteração do *for* mais externo. Pensando-se em desempenho, não deixamos que o protótipo analisa-se o interior de um *for* tipo *forall*. Caso fosse permitido, o sistema também detectaria que o *for* (*j)* também é do tipo *forall*.

Neste código exemplo, para que o *forall* seja detectado é necessário que a especificação semântica fornecida tome como relevante para a granularidade os comandos de atribuição. Caso contrário, o *for (k)* conterá um comando irrelevante, ou seja, ele também será irrelevante. O mesmo valerá para os comandos *for (j)* e *for (i)*.

O código paralelo *multithreaded* obtido respeitará a mesma lei de formação vista anteriormente (Mapeamento das Construções Paralelas – seção 3.6.1), ou seja, para cada subseção *begin/end* de uma seção *cobegin/coend* será gerada uma *thread*, da mesma maneira para cada iteração de um *forall*.

```
#include "funcs.h"

void main ()
{
   int i, j, k, tam;
   pthread_t tid [100];
   int countThreads = 0;
   int countAux;
   int countAux1;
   args_main * args [100];

   for (countAux = 0; countAux < 100; countAux ++)
        args [countAux] = (args_main *) malloc (sizeof (args_main));
   tam = 20;
   for (i = 0; i < tam; i = i + 1)
   {
       args [i] -> j = j;
       args [i] -> tam = tam;
       args [i] -> k = k;
       args [i] -> i = i;
(1)    pthread_create (& (tid [i]), NULL, func1, (void *) args [i]);
   }
   for (i = 0; i < tam; i = i + 1)
   {
(2)    pthread_join (tid [i], (void **) NULL);
   }
   for (countAux = 0; countAux < 100; countAux ++)
        free (args [countAux]);
}

(3)
void * func1 (void * p)
{
   pthread_t tid [100];
   int countThreads = 0;
```

```
   int countAux;
   args_main * args = p;
   for (args -> j = 0; args -> j < args -> tam; args -> j = args -> j + 1)
   {
      for (args -> k = 0; args -> k < args -> tam; args -> k = args -> k + 1)
      {
         C [args -> i] [args -> j] = C [args -> i] [args -> j] +
                                    (A [args -> i] [args -> k] * B [args -> k] [args -> j]);
      }
   }
}
```

Neste código, *(1)* cria as *threads* associadas a cada iteração; *(2)* sincroniza a execução, ou seja, espera pelo término das *threads* criadas, e; *(3)* é a função disparada pelas *threads* criadas. Os protótipos das funções e estruturas utilizadas por este código estão listados abaixo:

```
#include "stdio.h"
#include "pthread.h"
#include "errno.h"
#include "stdlib.h"
typedef struct
{
   int i, j, k, tam;
}
args_main;
void main ();
void * func1 (void * p);
```

4.2 – Quadratura Adaptativa

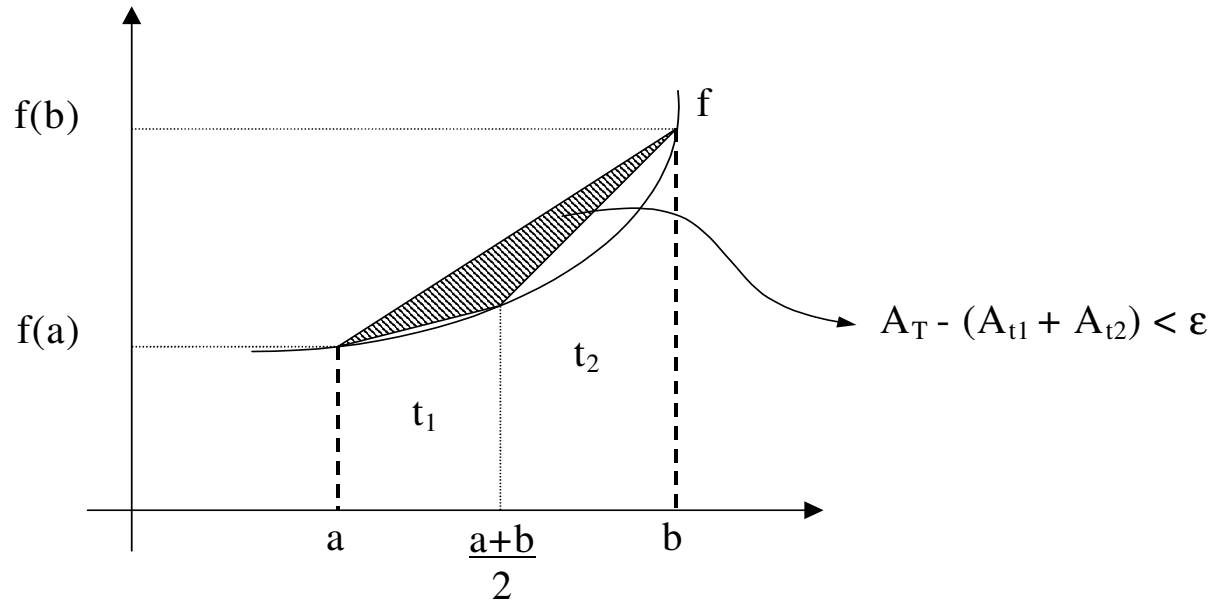


Figura 14 - Esquema do método dos trapézios

A quadratura adotada – método dos trapézios [RUL97] – calcula a integral de uma função f contínua em um intervalo T, de forma recursiva, decompondo-a no cálculo da integral de f nos dois subintervalos de T definidos pelo seu ponto médio. O procedimento atinge a base da recursão quando o módulo da diferença entre a área do trapézio definido pela função f no intervalo corrente e a soma das áreas dos trapézios definidos por f nos dois subintervalos for menor que um limite de tolerância para o erro (o trapézio definido por f no intervalo T=[a, b] é aquele formado pelos pontos extremos de T [(a,0), (b,0)] e pelos valores de f que estes extremos assumem [(a,f(a)) e (b,f(b))]). De acordo com a figura, a base da recursão é atingida quando o valor da área do triângulo sombreado for menor que o limite da tolerância para o erro. Neste momento, a área definida pelo intervalo corrente recebe o valor da soma das áreas definidas pelos respectivos subintervalos.

O código da implementação deste método terá a seguinte aparência:

```
void f ( float *retorno, float valor )
{
    float aux;

    aux = valor * valor;
    *retorno = aux;
}

void Area ( float *tamanho, float limEsq, float limDir )
{
    float Amaior, Aesq, Adir, ptoMedio;
    float FlimEsq, FlimDir, FPtoMedio;
    float valor1, valor2;

(1) ptoMedio = ( limEsq + limDir ) / 2;

(2) f ( &FlimEsq, limEsq );
(3) f ( &FlimDir, limDir );
(4) f ( &FPtoMedio, ptoMedio );

(5) Amaior = ((FlimEsq + FlimDir) * (limDir - limEsq)) / 2;
(6) Aesq = ((FlimEsq + FPtoMedio) * (ptoMedio - limEsq)) / 2;
(7) Adir = ((FlimDir + FPtoMedio) * (limDir - ptoMedio)) / 2;

(8) if ( ( Amaior - ( Aesq + Adir ) ) > 1 )
     {
(9)    Area ( &valor1, limEsq, ptoMedio );
(10)   Area ( &valor2, ptoMedio, limDir );
(11)   *tamanho = valor1 + valor2;
    }
    else
    {
(12)   *tamanho = Amaior;
    }
}
void main ()
{
    float AreaTrapezio;
    Area ( &AreaTrapezio, 10, 20 );
    printf ("Valor obtido: %f\n", AreaTrapezio);
}
```

Este código contém 3 funções: *main*, que inicia a execução do programa; *Area*, responsável pelas chamadas recursivas, que no caso base retornará a área ocupada por um determinado trapézio, e; *f* que é a mesma da ilustração, ou seja, sobre que função queremos realizar a integral (no caso dado, x^2). Neste exemplo estaremos dando enfoque apenas à função *Area*, pois as outras funções não apresentam instruções suficientes para que seja gerado algum tipo de paralelismo. O grafo de dependências de fluxo associado à função *Area* dada acima pode ser visualizado a seguir:

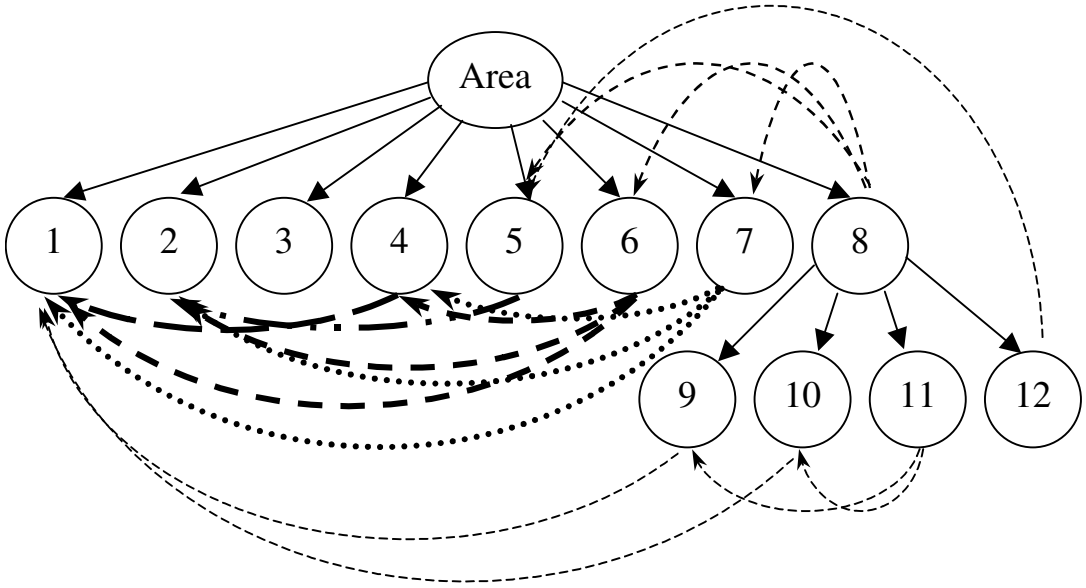


Figura 15 - Grafo de dependências de fluxo da função Area

À medida que códigos maiores são passados ao protótipo, surgem trechos paralelos não previstos (pelo usuário) no código intermediário. Na função *Area* acima, além das chamadas recursivas ((9) e (10) na figura), que são a parte fundamental do paralelismo, outros trechos também podem ser executados em paralelo: comandos (1) ‖ (2) ‖ (3); (4) ‖ (5); (6) ‖ (7) e as chamadas recursivas (9) ‖ (10), onde (x ‖ y) indica que x pode ser executado em paralelo com y.

Como vimos na seção 3.6.1, cada subseção gerada corresponde à uma *thread*. Logo, pensando-se em desempenho, é interessante que as subseções tenham computação

suficiente para compensar o disparo de uma *thread*. Para este exemplo, uma solução inicial seria tornar os nós do tipo atribuição como irrelevantes para a granularidade. Neste caso, o protótipo geraria as seguintes seções paralelas: (1 & 2) ‖ (3); (4 & 5); (6 & 7), e; (9) ‖ (10), onde (x & y) indica que x deve ser executado seqüencialmente com y. Se observarmos o código, podemos perceber que as intruções 1 e 2, executadas em paralelo com 3, também não contém computação que valha a criação de *threads*. Podemos então alterar o código intermediário de forma que estes comandos executem seqüencialmente, deixando a criação de *threads* apenas para as chamadas recursivas. O código intermediário resultante (para a função *Area*) pode ser visualizado a seguir:

```
void Area (float* tamanho, float limEsq, float limDir) {
        float Amaior, Aesq, Adir, ptoMedio;
        float FlimEsq, FlimDir, FPtoMedio;
        float valor1, valor2;
cobegin
   begin
(1)    ptoMedio=(limEsq+ limDir)/ 2;
(2)    f (& FlimEsq, limEsq);
(3)    f (& FlimDir, limDir);
   end
coend

cobegin
   begin
(4)     f (& FPtoMedio, ptoMedio);
(5)    Amaior=((FlimEsq+ FlimDir)*(limDir- limEsq))/ 2;
   end
coend

cobegin
   begin
(6)    Aesq=((FlimEsq+ FPtoMedio)*(ptoMedio- limEsq))/ 2;
(7)    Adir=((FlimDir+ FPtoMedio)*(limDir- ptoMedio))/ 2;
   end
coend

cobegin
   begin
```

```
(8)    if ((Amaior-(Aesq+ Adir))> 1) {
       cobegin
          begin
(9)           Area (& valor1, limEsq, ptoMedio);
          end
          begin
(10)          Area (& valor2, ptoMedio, limDir);
          end
       coend
       cobegin
          begin
(11)          * tamanho= valor1+ valor2;
          end
       coend
    } else {
       cobegin
          begin
(12)          * tamanho= Amaior;
          end
       coend
    }
   end
coend
}
```

Observe que o código intermediário tem uma versão mais simplificada, pois não falamos dos pontos de sincronismo. Como mostrado na seção 3.5, a simplificação deste código gera o seguinte código intermediário:

```
void Area (float* tamanho, float limEsq, float limDir) {
          float Amaior, Aesq, Adir, ptoMedio;
          float FlimEsq, FlimDir, FPtoMedio;
          float valor1, valor2;
(1) ptoMedio=(limEsq+ limDir)/ 2;
(2) f (& FlimEsq, limEsq);
(3) f (& FlimDir, limDir);
(4) f (& FPtoMedio, ptoMedio);
(5)Amaior=((FlimEsq+ FlimDir)*(limDir- limEsq))/ 2;
(6  Aesq=((FlimEsq+ FptoMedio)*(ptoMedio- limEsq))/ 2;
(7)Adir=((FlimDir+ FptoMedio)*(limDir- ptoMedio))/ 2;
```

```
(8)if ((Amaior-(Aesq+ Adir))> 1) {
    cobegin
        begin
(9)         Area (& valor1, limEsq, ptoMedio);
        end
        begin
(10)        Area (& valor2, ptoMedio, limDir);
        end
    coend
(11)  * tamanho= valor1+ valor2;
    } else {
(12)  * tamanho= Amaior;
    }
}
```

O código final *multithreaded* associado a este último exemplo de código intermediário pode ser visualizado a seguir:

```
#include "funcs.h"
void f (float * retorno, float valor)
{
   float aux;
   pthread_t tid [100];
   int countThreads = 0;
   int countAux;
   int countAux1;
   aux = valor * valor;
   * retorno = aux;
}
void Area (float * tamanho, float limEsq, float limDir)
{
   float Amaior, Aesq, Adir, ptoMedio;
   float FlimEsq, FlimDir, FPtoMedio;
   float valor1, valor2;
   pthread_t tid [100];
   int countThreads = 0;
   int countAux;
   int countAux1;
   args_Area * args;
   args = (args_Area *) malloc (sizeof (args_Area));
   ptoMedio = (limEsq + limDir) / 2;
```

```
    f (& FlimEsq, limEsq);
    f (& FlimDir, limDir);
    f (& FPtoMedio, ptoMedio);
    Amaior = ((FlimEsq + FlimDir) * (limDir - limEsq)) / 2;
    Aesq = ((FlimEsq + FPtoMedio) * (ptoMedio - limEsq)) / 2;
    Adir = ((FlimDir + FPtoMedio) * (limDir - ptoMedio)) / 2;
    if ((Amaior - (Aesq + Adir)) > 1)
    {
       args -> valor1 = valor1;
       args -> ptoMedio = ptoMedio;
       args -> limEsq = limEsq;
       pthread_create (& (tid [countThreads]), NULL, func1, (void *) args);
       countThreads ++;
       args -> valor2 = valor2;
       args -> ptoMedio = ptoMedio;
       args -> limDir = limDir;
       pthread_create (& (tid [countThreads]), NULL, func2, (void *) args);
       countThreads ++;
       for (countAux = countThreads - 2; countAux < countThreads; countAux ++)
             pthread_join (tid [countAux], (void **) NULL);
       valor1 = args -> valor1;
       ptoMedio = args -> ptoMedio;
       valor2 = args -> valor2;
       limEsq = args -> limEsq;
       limDir = args -> limDir;
       * tamanho = valor1 + valor2;
    }
    else
    {
       * tamanho = Amaior;
    }
    Amaior = args -> Amaior;
    Aesq = args -> Aesq;
    Adir = args -> Adir;
    valor1 = args -> valor1;
    ptoMedio = args -> ptoMedio;
    valor2 = args -> valor2;
    limEsq = args -> limEsq;
    limDir = args -> limDir;
    free (args);
}
void main ()
{
```

```
   float AreaTrapezio;
   pthread_t tid [100];
   int countThreads = 0;
   int countAux;
   int countAux1;
   Area (& AreaTrapezio, 10, 20);
   printf ("Valor obtido: %f\n", AreaTrapezio);
}
void * func1 (void * p)
{
   pthread_t tid [100];
   int countThreads = 0;
   int countAux;
   args_Area * args = p;
   Area (& args -> valor1, args -> limEsq, args -> ptoMedio);
}
void * func2 (void * p)
{
   pthread_t tid [100];
   int countThreads = 0;
   int countAux;
   args_Area * args = p;
   Area (& args -> valor2, args -> ptoMedio, args -> limDir);
}
```

O arquivo contendo os protótipos das funções e estruturas utilizadas também é mostrado abaixo:

```
#include "stdio.h"
#include "pthread.h"
#include "errno.h"
#include "stdlib.h"
typedef struct
{
   float Amaior, Aesq, Adir, ptoMedio;
   float FlimEsq, FlimDir, FPtoMedio;
   float valor1, valor2;
   float * tamanho;
   float limEsq;
   float limDir;
}
```

```
args_Area;
void f (float * retorno, float valor);
void Area (float * tamanho, float limEsq, float limDir);
void * func1 (void * p);
void * func2 (void * p);
void main ();
```

Capítulo 5 – Comparação com outros Sistemas

Neste capítulo, comparamos nossa arquitetura com alguns dos principais sistemas desenvolvidos.

Na comparação, buscamos discutir os itens que consideramos fundamentais para avaliarmos a qualidade de um sistema paralelizante, tais como, granularidade (específica ou geral), técnicas de paralelismo e otimização utilizadas, ambiente paralelo alvo, tipo de linguagem tratada, entre outros.

5.1 – CAPTools

CAPTools é uma ferramenta de paralelização semi-automática desenvolvida pelo Grupo de Processamento Paralelo (*Parallel Processing Group*) da Universidade de Greenwich [ICJ93]. CAPTools analisa código seqüencial e, com interação do usuário, gera código paralelo em FORTRAN77 muito similar ao original, mas com chamadas de comunicação inseridas e modificações no código que permitirão a execução num ambiente de processamento paralelo qualquer.

5.2 – CODE

CODE *(Computationally Oriented Display Environment)* é um sistema de programação paralelo visual desenvolvido na Universidade do Texas, o qual permite aos usuários a criação de um programa paralelo a partir da composição de programas sequenciais [NEB92]. O programa paralelo obtido é um grafo direcionado, onde os dados que fazem a comunicação entre os programas passam pelos arcos, que conectam os nós, os quais simbolizam os programas. Os programas seqüencias podem ser escritos em qualquer linguagem, e o sistema CODE produzirá programas paralelos para uma

variedade de arquiteturas, já que seu modelo é independente da arquitetura. CODE é projetado para paralelismos de granularidade grossa, indicando que os nós do grafo sejam computacionalmente grandes o suficiente. Estes nós são, normalmente, um conjunto de funções C, mas podem estar escritas em FORTRAN, C++, ou qualquer outra linguagem que possa ser "linkada" com C.

5.3 – Parallax

Iniciado em 1980, o projeto Parallax teve como objetivo a criação de um sistema paralelizante para transformação de programas, escritos em linguagens seqüenciais comuns como FORTRAN ou C, para suas formas paralelas, as quais funcionariam num ambiente distribuído. Parallax é um compilador fonte-a-fonte, que transforma programas escritos em linguagens com marcações especiais, em programas paralelos que podem ser "linkados" com qualquer sistema de troca de mensagens através de uma interface paralela universal. Para oferecer um poderoso tratamento à paralelismo a nível de tarefa, uma técnica de escalonamento estático avançada é utilizada. No estágio corrente, Parallax se concentra no escalonamento e geração de código paralelo. Portanto, nenhuma técnica de otimização significante é utilizada para expandir o paralelismo algoritmico das aplicações. O projeto Parallax é desenvolvido na Universidade Estadual de Moscow (Lomonosov).

5.4 – SUIF

SUIF (*Stanford University Intermediate Format*) é um compilador paralelizante que traduz programas seqüenciais em códigos paralelos para máquinas de memória compartilhada e distribuída [AAL95]. O compilador gera um programa SPMD que contém chamadas para bibliotecas de tempo de execução portáveis. Este projeto é desenvolvido na Universidade de Stanford.

5.5 – ParaScope

Este suporta programação paralela no nível de programas. A principal contribuição deste sistema é o entendimento de como os valores são compartilhados entre as diversas tarefas, num programa que executa sobre um ambiente compartilhado [CHH93]. Realiza análise interprocedural e otimização de programas Fortran. Contém um editor inteligente que fornece informação (ao programador) sobre como está sendo feito o compartilhamento de memória entre iterações em um "loop", auxiliando o programador na tomada de decisões. Este projeto é parte do Sistema D (D System), desenvolvido na *Rice University*, que contém outros projetos, como linguagens, editores, e etc.

5.6 - Parafrase-2

Parafrase-2 é um compilador "paralelizador/vetorizador" implementado como um reestruturador de código fonte-a-fonte [PGH89]. Contém "front-ends" para Fortran e C, e passos de análise, transformação e geração de código.

5.7 – ADAPTOR

ADAPTOR (*Automatic DAta Parallelism TranslaTOR*) é um sistema de domínio público que oferece compilação de HPF (*High Performance Fortran*) utilizando paralelismo de dados para computação paralela em máquinas de memória distribuída [BRZ94]. Através de transformações fonte-a-fonte, este sistema traduz programas de dados paralelos em seus equivalentes SPMD (*single program, multiple data*). Este projeto é desenvolvido pelo GMD (*German National Research Center for Information Technology*).

5.8 – FPT

FPT (*Fortran Parallel Transformer*) é um ambiente de programação paralelo que, a partir de programas seqüenciais Fortran, gera código para Fortran paralelo MP, múltiplas *threads*, PVM e grafo de tarefas [HFQ98]. FPT trabalha de forma automática ou interativa e pode paralelizar programas para arquiteturas do tipo SIMD e MIMD. Este projeto é desenvolvido pelo Grupo de Sistemas de Informação Paralelo da Universidade de Ghent.

5.9 – Quadro Comparativo

Característica / Sistema	Interação	Técnicas Aplicadas	Linguagem Alvo	Legibilidade Código	Granular.	Arquitetura Memória
CAPTools	sim	muitas	Fortran77	boa	-	distribuída
CODE	sim	poucas[1]	qualquer[2]	-	alta	compartilhada distribuída
Parallax	-	poucas	qualquer[3]	-	-	distribuída
SUIF	-	muitas	C, Fortran	-	-	compartilhada distribuída
ParaScope	sim	muitas	Fortran	-	alta	distribuída
Parafrase-2	-	muitas	C, Fortran	-	-	compartilhada distribuída
ADAPTOR	-	muitas	Fortran	boa	-	distribuída
FPT	sim	muitas	Fortran	-	-	compartilhada distribuída
Nosso	sim	poucas	qualquer[4]	boa	qualquer[5]	compartilhada

[1] As técnicas são aplicadas ao grafo de programas, não a cada programa (nó do grafo) em particular.
[2] Quaisquer linguagens que "linkem" com C.
[3] Linguagens com marcações especiais.
[4] Quaisquer, desde que seja fornecida sua especificação semântica.
[5] Especificada pelo usuário.

Capítulo 6 - Conclusão

Como comentado na Seção 1.1 (“Motivação”), procuramos projetar uma arquitetura que tirasse do programador a responsabilidade total de magicamente “criar” código paralelo, suavizando o árduo trabalho de se programar utilizando paralelismo, da mesma forma que fazem as linguagens de alto de nível com respeito às linguagens de máquina.

Além da geração automática de código paralelo, que é inerente à proposta principal deste trabalho, também podemos ressaltar como grande resultado o fato de conseguirmos recuperar o código com sintaxe no nível do código seqüencial original. Este código, que após a geração contém construções paralelas do tipo *cobegin/coend* e *forall,* pode ser alterado pelo usuário, desde que este respeite a sintaxe entendida pelo módulo que faz o pós-processamento (instanciação do protótipo à uma arquitetura específica).

A modularização de nosso sistema possibilitou uma série de vantagens. Do ponto de vista da arquitetura, podemos encarar os módulos até a fase do pós-processamento como independentes da máquina alvo. Ou seja, deixamos para o módulo, que instancia o protótipo a uma determinada arquitetura, todo conhecimento necessário para que esta transformação seja feita. Obviamente, não está sendo discutida a facilidade de se implementar esta transformação, mas sim, de destacar a independência do sistema com relação à uma arquitetura (até o pós-processamento).

A genericidade, com respeito às linguagens a serem tratadas, com certeza é um fator de discussão. Por exemplo, da tabela ilustrada na Seção 4.9, onde comparamos nosso sistema com similares, observamos que todos os sistemas que se proporam a tratar a linguagem de entrada de forma genérica tiveram que abdicar de algum fator. Observamos, ao longo do trabalho, que certas particularidades não puderam ser tratadas pela nossa arquitetura. Por exemplo, na Seção 3.4, falamos que não conseguimos avaliar expressões de forma a saber se 2 (duas) expressões sempre têm o mesmo valor ou não.

Na verdade, o que também foi sugerido naquela seção, poderíamos, além dos identificadores, mapear também as funções da linguagem tratada na meta-linguagem. Assim teríamos capacidade de fazer tal comparação, já que o conjunto de funções da meta-linguagem é finito, diferente da linguagem alvo, já que esta pode ser qualquer. Independente dos problemas encontrados e/ou identificados, consideramos válida e audaciosa a proposta genérica, o que contribuiu bastante para o enriquecimento do trabalho.

A flexibilidade na granularidade do paralelismo que se deseja obter também é algo a ser citado. Apesar de não termos feito vastos testes como por exemplo, granularidade de programas, acreditamos que a idéia é bastante razoável e de grande valia para os pesquisadores de compiladores paralelizantes.

Falando especificamente do protótipo, uma dificuldade encontrada foi o tratamento de erros retornados pela ferramenta de transformação (TXL), já que esta retorna informações puramente sintáticas e na linguagem da própria ferramenta. Esta dificuldade inviabilizou a decodificação do erro para informação ao usuário de maneira mais amigável. Uma das situações onde isto ocorre é quando fazemos a geração de um protótipo cuja especificação da linguagem alvo está incorreta sintaticamente.

Apesar destas dificuldades, o protótipo implementado, o qual processa um subconjunto da linguagem C, é utilizável. A única ressalva é quanto aos pseudônimos causados por variáveis que são referências para áreas de memória (ponteiros). Dependências associadas a este tipo de variável não foram tratadas dada sua complexidade.

Indubitavelmente, muita pesquisa ainda precisa ser feita para que possamos explorar todas as vantagens de processos concorrentes. Esperamos que este trabalho tenha sido um passo, reconhecidamente pequeno, mas na direção certa do desenvolvimento de ferramentas paralelas.

Finalizando, 2 pensamentos que guardei quando comecei este trabalho e que mostram o quanto ainda temos a percorrer. Isto devido à data dos pensamentos e ao quanto conseguimos obter com paralelismo até hoje:

``...the advent of parallel programming may do something to revive the pioneering spirit in programming, which seems at the present to be degenerating into a rather dull and routine occupation."---S. Gill, 1958.

``For over a decade prophets have voiced the contention that the organisation of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution."---G.M. Amdahl, 1967.

Apêndice A - Descrição do Formato do Analisador

O analisador a ser gerado tem a função de percorrer o código dado como entrada e detectar as dependências existentes neste. Estas dependências servirão de entrada para outro módulo, que terá a incumbência de alocar o grafo de dependências, de forma que este possa ser pecorrido e manipulado para a detecção de paralelismo.

Este analisador é implementado em TXL, pois precisamos reconhecer padrões (códigos numa dada linguagem).

Em termos gerais, para 1 (uma) dada regra gramatical

```
Label1:  RuleName1= LeftSide1 |
Label2:             LeftSide2 |
…
LabelN:             LeftSideN ;
```

basicamente, N+1 funções são geradas: func_RuleName1, func_Label1, func_Label2, …, func_LabelN. As funções do tipo func_RuleNameX[11] tem por finalidade, dada uma estrutura sintática, saber que lado-direito é compatível a esta e dar o tratamento semântico correspondente (retornar as dependências). Os testes para cada possível lado-direito são realizados através das chamadas das funções func_LabelX. Logo, as funções do tipo func_RuleNameK conterão chamadas para todas as funções do tipo func_LabelX que façam parte da regra RuleNameK.

Mostraremos, através de um pequeno exemplo, qual é o esquema criado para a geração do analisador, que é a saída deste primeiro módulo. Para isto, utilizaremos o exemplo abaixo, que é a mesma gramática de expressões vista na Seção 3.7.1 (um pouco mais detalhada).

[11] Cada regra da gramática de entrada terá sua função func_RuleNameX associada.

```
EXPRESSION: EPSILON = EPSILON OMICROM EPSILON |
FATOR:              '(' EPSILON ')' |
ID:                 NI |
NUM:                PI ;
IDENT:              NI = id ;
NUMERO:             PI = number ;
```

Para o identificador da regra (1o. “EPSILON”), é criada uma função (func_EPSILON) a qual conterá chamadas para as funções que tratarão cada possível lado-direito. Estas serão identificadas pela palavra “func” concatenada com o rótulo da regra: func_EXPRESSION, func_FATOR, func_ID, etc, e tratarão seus respectivos lados-direito, chamando as funções correspondentes no caso de não-terminais e dando o tratamento semântico adequado no caso de terminais. Esta função teria a seguinte aparência:

```
function func_EPSILON PROG [EPSILON]
                          SEM [repeat semantic_equation]
    replace [repeat anyToken]
        INI [repeat anyToken]
    construct  PROG1 [synt_structure]          (1)
        PROG
    construct  INI2 [repeat anyToken]          (2)
        _ [func_EXPRESSION  PROG1  SEM]
          [func_FATOR  PROG1  SEM ]
          [func_ID  PROG1  SEM]
          [func_NUM  PROG1  SEM ]
          [func_IDENT  PROG1  SEM]
          [func_NUMERO  PROG1  SEM ]
    by  INI2
end function
```

, onde, daqui para frente, os termos em *itálico* representam elementos da descrição semântica dada como entrada e os termos em **negrito** são palavras reservadas de TXL, e; (1) é a construção da variável (código de entrada) a ser passada para as funções a serem chamadas e (2) são as chamadas das funções que tratarão os possíveis lados-direito do não-terminal EPSILON.

Observaremos agora, como é a geração destas funções que tratam o lado-direito de uma equação semântica. Para isto, veremos como a função associada ao rótulo NUM (func_NUM) é construída, analisando passo a passo cada construção TXL. A função gerada para o pequeno exemplo é:

```
function func_NUM  PROG [synt_structure]
            SEM [repeat semantic_equation]
        replace [repeat anyToken]
        construct  INI [repeat anyToken]                    (1)
        deconstruct  PROG                                   (2)
            var_PI0  [PI]
        deconstruct  *  [semantic_equation] SEM             (3)
            _ [id] '[[ NUM ']] '=  RS [rightSide] ';
        construct  RIGHTSIDE [repeat idOrCharlit]           (4)
            'PI
        construct  RIGHTSIDE_NUMBER [repeat  idOrCharlit]   (5)
            'PI0
        construct  PROG1 [repeat synt_structure]            (6)
            var_PI0
        construct  DEPENDS [repeat semanticValue]           (7)
            _ [solveSyntSemRightSide RIGHTSIDE
                                     RIGHTSIDE_NUMBER
                                     PROG1
                                     SEM]
```

by

INI [semanticFunction DEPENDS RS] (8)

end function

Nesta função, (1) constrói apenas uma variável auxiliar a ser usada posteriormente; (2) é um guarda que testa se o tipo analisado é o lado direito de NUM; (3) busca a equação semântica associada à meta-variável NUM; (4) é a lista de não-terminais da regra sintática a ser passada para a função "solveSyntSemRightSide", que resolve o lado-direito chamando as funções correspondentes aos não-terminais; (5) é a lista dos mesmos não-terminais indexados; (6) constrói uma lista de variáveis que contêm os elementos sintáticos obtidos em (2); (7) chama a função "solveSyntSemRightSide" que gera uma lista de tuplas (não-terminal, valor do não-terminal) a partir da regra corrente, e; (8) que chama a função "semanticFunction", a qual utiliza as tuplas geradas (DEPENDS) e a informação do tipo de regra aplicada (RS) para gerar as dependências, caso existam.

Como a formatação do lado-direito de uma Equação Semântica é pré-definida[12], o que viabilizou a genericidade da arquitetura, as funções "solveSyntSemRightSide" e "semanticFunction" puderam ser definidas independentemente da linguagem e código analisados[13].

[12] Ver Seção 3.7.1

[13] Estas funções tem seu comportamento precisamente descrito no Projeto Final de Programação, desenvolvido pelo mesmo candidato como um dos pré-requisitos para obtenção do título de Mestrado.

Apêndice B - Descrição do Formato Textual das Dependências

Este formato, dentre várias outras particularidades (em relação ao nosso protótipo) citadas neste texto, são dispensáveis para o entendimento da arquitetura. Esta formatação poderia não existir caso o módulo que detecta das dependências também fizesse a manipulação do grafo de dependências. Em nosso protótipo, esta divisão nos pareceu mais interessante, visto que utilizamos ferramentas diferentes neste módulos.

Esta formatação é bastante similar às representações de expressões na forma parentizada. Desta maneira, conseguimos, com facilidade, percorrer a expressão parentizada e gerar a estrutura do grafo associado.

Como a arquitetura prevê a recuperação do código no nível de abstração do código de entrada, informações além do aninhamento sintático do código (dado pela construção da expressão parentizada) precisam ser passadas. Em nosso protótipo, passamos informações como o identificador da regra que originou a construção sintática, trechos do código original além, é claro, dos operadores que indicam as dependências na construção tratada.

As seguintes estruturas podem aparecer no arquivo de dependências (formato BNF):

<Atrib> ::= **“[”**<id>**“]” “(”** <id> **“<-”** {<id>} **“{”**<Code>**“}” “)”**

Este é o formato de uma atribuição, sendo que o identificador antes do símbolo **“<-”** é o lado esquerdo de uma atribuição, e os identificadores da repetição do ramo direito são os que constituem a expressão atribuída (o lado esquerdo dependerá dos elementos do lado direito); o identificador entre colchetes (**[]**) é a meta-variável (Ver

Regras de Produção Abstrata na Seção 3.7.1) que identifica o formato da atribuição dada na linguagem de entrada; <Code> é o código correspondente à dependência tratada.

Apesar de apresentarmos esta construção diretamente como a representação da atribuição, esta pode ser usada para outros comandos. Para isto, basta que as dependências neste comando sejam similares as dependências encontradas num comando de atribuição, ou seja, que as porções que leem e escrevem valores estejam explicitadas no comando.

<If> ::= **"["**<id>**"]" "(""["**<id>**"]"** {<Code>} **"->"**
"["<id>**"]"** { <Atrib> | <If> }
[**","** **"["**<id>**"]"** { <Atrib> | <If> }]
")"

Este, adequado a um condicional, criará uma dependência entre os identificadores da expressão condicional (anteriores ao símbolo **"->"**) e os comandos do ramo *if* e do ramo *else* (posteriores ao símbolo **"->"**). O primeiro identificador (entre colchetes **[]**) é uma meta-variável que tem a mesma função da definida na atribuição: identificar que regra sintática gerou o comando tratado. Dentro dos parênteses (()) encontramos 3 identificadores entre colchetes, os quais me fornecem os não-terminais que compõem o condicional.

A observação feita na estrutura da atribuição, com respeito ao seu uso de forma geral, também é válida para esta estrutura. Ou seja, caso tenhamos outras construções sintáticas que tenham informações de dependência dispostas da mesma maneira que no condicional, podemos reutilizá-la.

Apêndice C – Especificação Semântica da linguagem C (protótipo)

Neste apêndice, mostramos como ficou a especificação semântica do subconjunto da linguagem C processada pelo nosso protótipo.

```
PSI: Prog          ( programas  )
EPSILON: Expr      ( expressoes )
DELTA: Bloco       ( blocos )
SIGMA: Stmt        ( comandos )
NI: Id             ( identificadores )
PI: Num            ( numeros )
KAPPA: Comp        ( comparacao )
BETA: OpRel        ( relacao )
ALFA: OpSoma       ( soma )

@@

PROGRAM:    PSI   = OMEGA ;
FUNCS1:     OMEGA    = OMEGA1 OMEGA |
FUNCS2:              OMEGA1 ;
VARS1:      OMEGA1   = TYPE NI '(' PARAM ')' '{' LISTVARS SIGMA '}' |
VARS2:               TYPE NI '(' ')' DELTA |
VARS3:               TYPE NI '(' PARAM ')' DELTA |
VARS4:               TYPE NI '(' ')' '{' LISTVARS SIGMA '}' ;
PARS1:      PARAM    = TYPE NI ',' PARAM |
PARS2:               TYPE NI ;
ARGS1:      ARGS   = NI ',' ARGS |
ARGS2:             NI |
```

```
        ARGS3:                  PI ',' ARGS |
        ARGS4:                  PI ;
        LISTVARS1:  LISTVARS    = TYPE LIST LISTVARS |
        LISTVARS2:              TYPE LIST ;
        LISTVARS3:  LIST = NI ',' LIST |
        LISTVARS4:          NI ';' ;
        TYPES0:     TYPE    = NI ;
        BLOCO:      DELTA = '{' SIGMA '}' ;
        STMT_LIST1: SIGMA = SIGMA1 SIGMA |
        STMT_LIST2:         SIGMA1 ;
        ASSIGN:     SIGMA1  = NI '=' EPSILON ';' |
        CALL:               NI '(' ARGS ')' ';' |
        CALL1:              NI '(' ')' ';' |
        IF:                 'if' '(' KAPPA ')' DELTA |
        IF_ELSE:            'if' '(' KAPPA ')' DELTA 'else' DELTA |
        FOR:                'for' '(' NI '=' EPSILON ';' KAPPA ';' NI '=' EPSILON ')'
DELTA ;
        COMP:       KAPPA   = EPSILON RO EPSILON |
        COMP1:              EPSILON ;
        EXPRESSION: EPSILON = EPSILON OMICROM EPSILON |
        FATOR:                '(' EPSILON ')' |
        ID:                   NI |
        NUMERO:               PI ;
        MAIS:           OMICROM = '+' |
        MENOS:                    '-' |
        VEZES:                    '*' |
        DIVIDE:                   '/' ;
        IGUAL:          RO      = '==' |
        DIFERENTE:              '!=' |
        MENOR:                  '<' |
        MENOR_OU_IGUAL:         '<=' |
```

```
MAIOR:                  '>' |
MAIOR_OU_IGUAL:   '>=' ;
IDENT:      NI    = id |
IDENT1:           id '[' EPSILON ']' '[' EPSILON ']' |
IDENT2:           '*' id |
IDENT3:           '*' id '[' EPSILON ']' |
IDENT4:           '&' id |
IDENT5:           '&' id '[' EPSILON ']' |
IDENT6:           stringlit |
IDENT7:           id '[' EPSILON ']' ;
NUM:        PI    = number ;

@@

lambda: Graph = Id* ;

@@

M: Prog -> _ => Graph;
B: Bloco -> Graph => Graph;
S: Stmt -> Graph => Graph;
E: Expr -> Graph => Graph;
C: Comp -> Graph => Graph;
O: Opr  -> Graph => Graph;
R: Rel  -> Graph => Graph;

@@

M [[ PROGRAM ]] = S [[ OMEGA0 ]] ;
D [[ FUNCS1 ]] = S [[ OMEGA0 ]] (S [[ OMEGA10 ]]) ;
D [[ FUNCS2 ]] = S [[ OMEGA10 ]] ;
```

D [[VARS1]] = NI0 < D [[PARAM0]] > <-> B [[SIGMA0]] ;

D [[VARS2]] = NI0 <-> B [[DELTA0]] ;

D [[VARS3]] = NI0 < D [[PARAM0]] > <-> B [[DELTA0]] ;

D [[VARS4]] = NI0 <-> B [[SIGMA0]] ;

D [[PARS1]] = forward ;

D [[PARS2]] = forward ;

D [[ARGS1]] = forward ;

D [[ARGS2]] = forward ;

D [[ARGS3]] = forward ;

D [[ARGS4]] = forward ;

D [[LISTVARS1]] = forward ;

D [[LISTVARS2]] = forward ;

D [[LISTVARS3]] = forward ;

D [[LISTVARS4]] = forward ;

D [[TYPES0]] = forward ;

B [[BLOCO]] = S [[SIGMA0]] ;

S [[STMT_LIST1]] = S [[SIGMA0]] (S [[SIGMA10]]) ;

S [[STMT_LIST2]] = S [[SIGMA10]] ;

S [[ASSIGN]] = >>> NI0 <- E [[EPSILON0]] <<< ;

S [[CALL]] = call NI0 < P [[ARGS0]] > ;

S [[CALL1]] = call NI0 ;

S [[IF]] = C [[KAPPA0]] -> B [[DELTA0]] ;

S [[IF_ELSE]] = C [[KAPPA0]] -> B [[DELTA0]], B [[DELTA1]] ;

S [[FOR]] = NI0 E [[EPSILON0]] C [[KAPPA0]] NI1 E [[EPSILON1]] -> B [[DELTA0]] ;

E [[EXPRESSION]] = forward ;

E [[ID]] = forward ;

E [[NUMERO]] = forward ;

E [[IDENT]] = forward ;

E [[IDENT1]] = ref id0 [EPSILON0 EPSILON1] ;

E [[IDENT2]] = forward ;

E [[IDENT3]] = ref id0 [EPSILON0] ;

E [[IDENT4]] = forward ;

E [[IDENT5]] = ref id0 [EPSILON0] ;

E [[IDENT6]] = forward ;

E [[IDENT7]] = ref id0 [EPSILON0] ;

E [[NUM]] = forward ;

E [[FATOR]] = forward ;

C [[COMP]] = forward ;

C [[COMP1]] = E [[EPSILON0]] ;

O [[MAIS]] = nil ;

O [[MENOS]] = nil ;

O [[VEZES]] = nil ;

O [[DIVIDE]] = nil ;

R [[IGUAL]] = nil ;

R [[DIFERENTE]] = nil ;

R [[MENOR]] = nil ;

R [[MENOR_OU_IGUAL]] = nil ;

R [[MAIOR]] = nil ;

R [[MAIOR_OU_IGUAL]] = nil ;

Referências Bibliográficas

[AAL95] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng, "The SUIF Compiler for Scalable Parallel Machines", Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, February, 1995.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques and Tools. Addison Wesley, 1986.

[BAS96] Baker, Louis, Smith, Bradley J., "Parallel Programming", McGraw-Hill, 1996.

[BCFH89] Burke, Michael, Cytron, Ron, Ferrante, Jeanne, Hsieh, Wilson. "Automatic Generation of Nested, Fork-Join Parallelism", The Journal of Supercomputing 3, 71-88 (1989).

[BRZ94] Brandes, Thomas; Zimmermann, Falk, "ADAPTOR - A Transformation Tool for HPF Programs", In: Programming Environments for Massively Parallel Distributed Systems, Birkhäuser Verlag, April 1994, pages 91-96.

[CF87b] Cytron, Ron, Ferrante, Jeanne, "What's in a name? –or- the value of renaming for parallelism detection and storage allocation", In Proceedings of the 1987 International Conference on Parallel Processing, pages 19-27, St. Charles, IL, August, 1987.

[CHH93] K. D. Cooper, M. W. Hall, R. Hood, K. Kennedy, K. McKinley, J. Mellor-Crummey, L. Torczon, and S. Warren, "The ParaScope Parallel Programming Environment", In Proceedings of the IEEE, pages 244-263, February 1993.

[GRA98] Gray, John Shapley, "Interprecess Communications in UNIX", Prentice Hall PTR, 1998.

[GUE95] Guedes, Luiz Carlos Castro, "Um Modelo Orientado a Objetos para Geração Automática de Compiladores", Tese de Doutorado, PUC-RJ, 1995.

[HFQ98] D'Hollander Erik H., Zhang Fubo, Wang Qi, "The Fortran parallel transformer and its programming environment", Journal of Information Sciences. Vol. 106, pp. 293-317. Elsevier Science Inc., Jul. 1998.

[ICJ93] C. S. Ierotheou, M. Cross, S. P. Johnson, P. F. Leggett. "CAPTools - An interactive toolkit for mapping CFD codes onto parallel architectures", Parallel CFD 93, North Holland, p251, 1993.

[KUC78] Kuck, D. J., "The Structure of Computers and Computations", volume 1, John Wiley and Sons, New York, 1978.

[LEW96] Lewis, Bil, Berg, Daniel J. "Threads Primer: A Guide do Multithreaded Programming, Sun Microsystems, Inc. 1996.

[MHPCC] MAUI HIGH PERFORMANCE COMPUTING CENTER, 1995 (http://www.uni-karlsruhe.de/Uni/RZ/Hardware/SP2/Workshop.mhpcc/ parallel-intro/ParallelIntro.html#paradigms

[MOR94] Morse, H. Stephen, "Practical Parallel Computing", AP Professional, 1994.

[NEB92] P. Newton, J. C. Browne, "The CODE 2.0 Graphical Parallel Programming Language", Proc. ACM International Conference on Supercomputing, July, 1992.

[PAG81] Pagan, Frank G., "Formal specification of programming languages", Prentice-Hall, INC., Englewood Cliffs, New Jersey.

[PBJ91] Pingali, K., Beck, M., Johnson, R., Moudgill, M., Stodghill, P., "Dependence Flow Graphs: an Algebraic Approach to Program Dependencies", POPL, 1991.

[PGH89] Constantine Polychronopoulos, Milind B. Girkar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, Dale A. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors", Proceedings of the International Conference on Parallel Processing, St. Charles IL, August 1989, pp. II39-48; also in International Journal of High Speed Computing, Vol. 1, No. 1, 1989.

[RUL97] M. A. G. Ruggiero, V. L. R. Lopes, Cálculo Numérico: Aspectos Teóricos e Computacionais, Makron Books, 1997.

[TXL95] Cordy, James R., Carmichael, Ian H., Halliday, Russel, "The TXL Programming Language", Version 8, Queen's University, Kingston, Ontario, Canada.

[WCH88] Hsieh, Wilson Cheng-Yi, “Extracting Parallelism from Sequential Programs”, MSc. Thesis, Massachusetts Institute of Technology, 1988.

[WOL96] Wolfe, Michael Joseph, “High Performance Compilers for Parallel Computing”, Oregon Graduate Institute of Science & Technology, Addison-Wesley Publishing Company.