Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Tópicos em Compiladores
Otimização de Código

Série de Seminários 1999-1

Relatório Técnico LLP006/99

Editado por

Mariza Andrade da Silva Bigonha

Belo Horizonte - setembro de 1991

Caixa Postal, 702

30.161 - Belo Horizonte - MG

10 de setembro de 1999

Sumário

Apresentação

Este Relatório Técnico do Laboratório de Linguagens de Programação apresenta o texto de dez seminários apresentados no primeiro semestre de 1999 pelos alunos da disciplina *Tópicos em Compiladores - ênfase: Otitimização de Código* do Curso de Pós-Graduação em Ciência da Computação da UFMG. Esta coletânea tem por objetivo divulgar uma parte das tarefas desenvolvidas pelos alunos da disciplina e também mostrar parte da pesquisa que está sendo realizada no Laborátorio de Linguagens de Programação.

A ênfase dos cinco primeiros trabalhos está centrada na parte de análise de fluxo de controle, de dados, análise de dependência e alias e foi baseada principalmente no livro texto: "Advanced Compiler Design & Implementation" de Steven S. Muchnick, edição 1997.

Os outros cinco trabalhos são artigos na área de Implementação de Linguagens com ênfase em otimização de código. Especificamente, o sexto artigo trata da *Avaliação Parcial de Programas*. A avaliação parcial de um programa P com duas entradas i_1 e i_2 produz em relação a entrada i_1 um novo programa P_{i_1} denominado especializado ou residual. O novo programa P_{i_1} quando executado sobre a entrada restante i_2 produz o mesmo resultado que a execução de P sobre ambas as entradas. O objetivo principal da avaliação parcial é o ganho em eficiência. O raciocínio por trás desta abordagem reside no fato de que se parte dos dados de entrada de um programa é conhecida, as estruturas do programa que dependam apenas dessa parte podem ser previamente computadas. Conseqüentemente, o programa especializado conterá apenas o código necessário para processar os dados ainda não conhecidos.

O sétimo artigo, *Compilação e Otimização de Código Machĩna para Código C** aborda o problema de compilação e otimização de código escrito na linguagem Machĩna para código C. Machĩna é uma linguagem baseada em Máquinas de Estado Abstratas e possui diversas construções de alto nível. A maior dificuldade na compilação de código Machĩna para C está na tradução do modelo de execução de Machĩna, que é essencialmente paralelo, para o modelo seqüencial de C.

O oitavo artigo, *Coleta de Lixo* faz uma revisão dos processos que permitem o gerenciamento automático de memória de tal forma que entidades alocadas dinâmicamente são liberadas, quando não mais necessárias, sem a intervenção do programador. Este tópico é de suma importância, especialmente na implementação de linguagens funcionais.

O nono artigo, *Estratégias para Otimização de Código Móvel* mostra as principais estratégias de otimização usadas em compiladores Java propostas recentemente com o objetivo de produzir um código portável *bytecode* mais eficiente. Ênfase é dada a três compiladores: Marmot da Microsoft Research, Cream da Universidade de Aarhus e Briki da Universidade de Rochester.

O décimo artigo, *Aspectos Relevantes da Implementação de Abstrações de Controle Paralelo em Linguagens Orientadas por Objeto* trata de um dos maiores desafios na paralelização de aplicações que é a otimização de código paralelizado de forma a melhor explorar os recursos que o ambiente de execução provê. Este desafio é uma constante durante o ciclo de vida de uma aplicação paralelizada, seja por mudanças nos parâmetros de entrada ou nos recursos disponíveis. Uma estratégia para facilitar a paralelização de aplicação é utilizar mecanismos, tais como controle de abstração, que permitam expressar as oportunidades de paralelismo dessas aplicações, sem que se defina a sua implementação, o que as torna independentes de plataforma de execução.

Faz parte também deste relatório o artigo *Processador para o Bytecode de Java* que apresenta os principais aspectos do código *bytecode*, faz uma análise do conjunto de instruções

da máquina virtual Java com as do processador Risc e mostra as mudanças no *bytecode* necessárias no projeto de um processador para Java.

Todos os artigos terminam com uma lista de referências que o leitor interessado poderá consultar para aprofundar nos diferentes tópicos apresentados.

Gostaria de agradecer a todos os alunos pelo cuidado e carinho com que prepararam os trabalhos apresentados.

Mariza Andrade da Silva Bigonha

imediatamente antes do seu header. Para garantir que tais movimentações possam sempre ser efetuadas introduziremos o conceito de preheader.

Um preheader é um novo bloco (inicialmente vazio) que é conectado ao header do loop de tal forma que todas as arestas externas ao loop incidentes no header são direcionadas ao preheader. Uma nova aresta (única) conecta o preheader ao antigo header. A Figura 9 mostra um exemplo de um loop antes e após a inserção de um preheader.

Introdução a Análise de Fluxo de Controle

Mark Alan Junho Song

1 Introdução

A otimização de código pode ser vista como o processo de se gerar códigos que utilizem de forma eficiente os recursos disponíveis levando em consideração: o uso de registradores, utilização de instruções adequadas, escalonamento de uma seqüência de instruções etc. Um código intermediário é, em geral, inadequado para o processo de otimização por não fornecer informações explícitas a respeito do fluxo de controle e fluxo de dados de um programa. Note, por exemplo, que dado o programa:

```
int fib(int m)
{
  int f0 = 0, f1 = 1, f2, i;
  if (m <= 1)
     return m;
  else {
     for (i = 2; i <= m; i++) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
     }
     return f2;
  }
}
```

e a seguinte representação intermediaria:

```
     receive m(val)
     f0 <- 0
     f1 <- 1
     if m <= 1 goto L3
     i  <- 2
L1:  if i <= m goto L2
        return f2
L2:  f2 <- f0 + f1
     f0 <- f1
     f1 <- f2
     i  <- i + 1
     goto L1
L3:  return m
```

Necessitamos de uma representação mais adequada que colete informações do programa intermediário exibindo-as explicitamente. Tais informações podem ser representadas eficientemente por um Grafo de Fluxo (que é um grafo direcionado) onde:

- os nós especificam uma computação do código intermediário e,

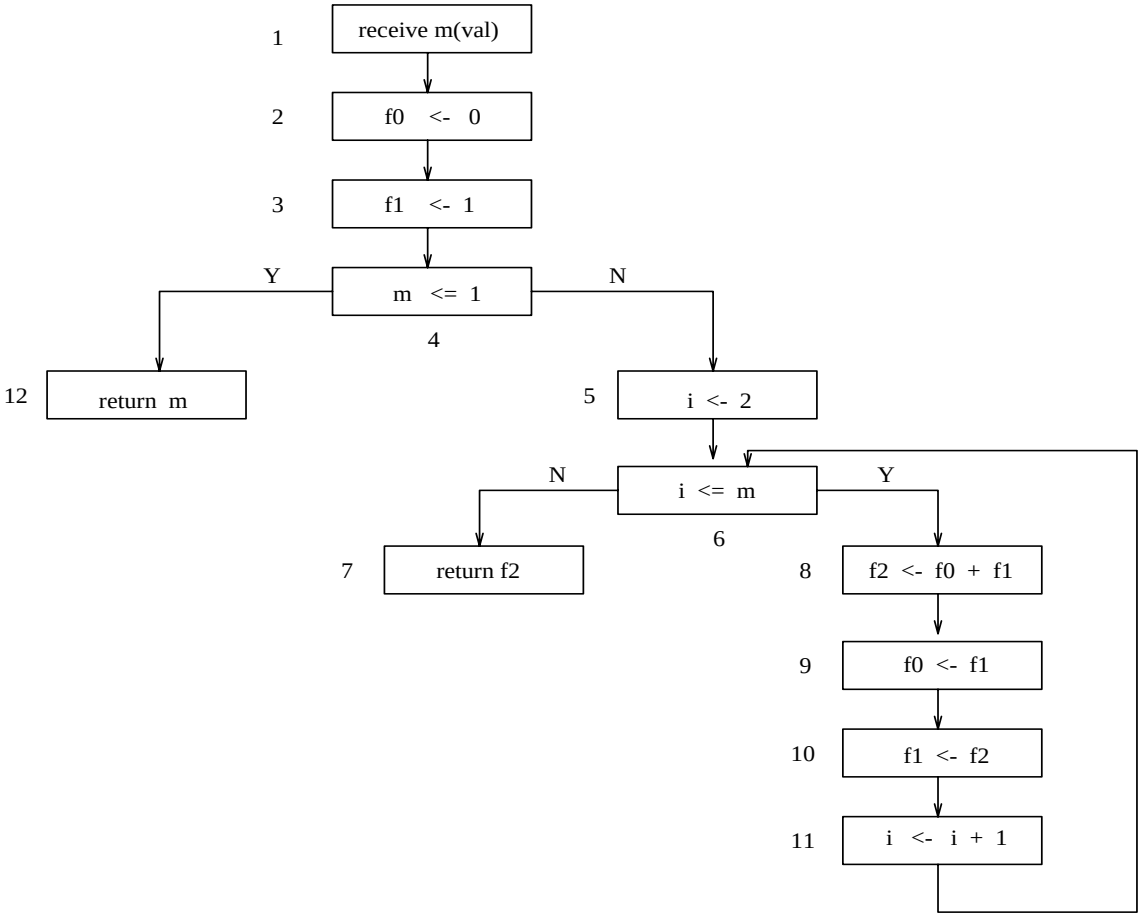


Figura 1: Um Grafo de Fluxo para o programa anterior.

- as arestas definem o fluxo de controle entre os mesmos.

Uma seqüência de comandos será identificada por um grafo contendo:

- uma única entrada associada a seqüência,
- os blocos básicos especificados como nós e,
- um único término para a execução da mesma.

2 Blocos Básicos

As abordagens de Análise de Fluxo de Controle partem da determinação de um bloco básico de uma rotina para a construção de seu grafo de fluxo. Formalmente um bloco básico é uma seqüência máxima de instruções que pode ser alcançada pela primeira instrução (entrada do

bloco) e encerrada pela execução da última instrução desta seqüência (saída do bloco). A primeira instrução do bloco, também denominada *Líder*, será definida como:

- o ponto de entrada da rotina,
- a instrução alvo de um desvio ou,
- a instrução que segue imediatamente um desvio.

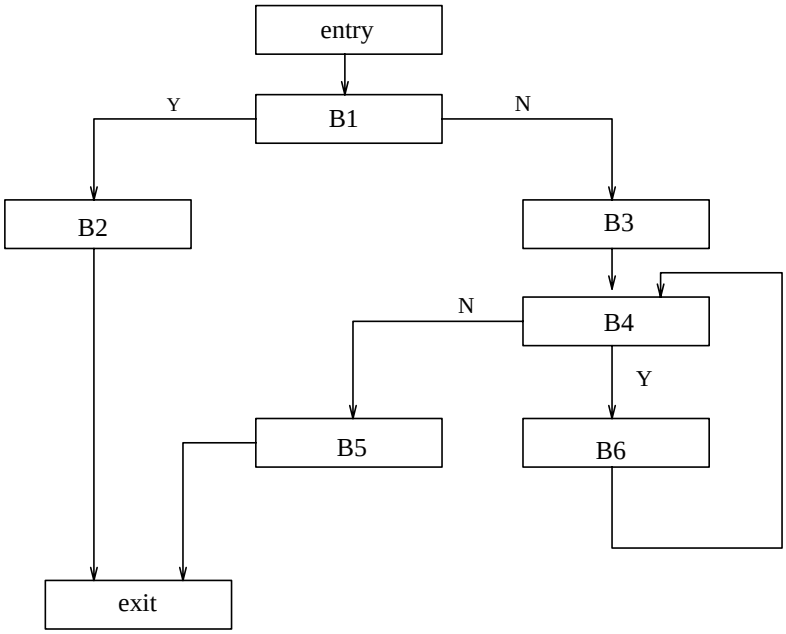


Figura 2: Grafo de Fluxo com blocos básicos.

Um bloco básico contém, desta forma, um líder e todas as instruções que o seguem até que um novo líder ou o fim da rotina seja encontrado na seqüência. Assumiremos, para efeito de definições, que um grafo de fluxo G = $< N, E >$ especifica um conjunto N de nós e um conjunto E de arestas (E $\subseteq$ N x N) onde *entry, exit* $\in N$. Segue abaixo, um algoritmo que determina o grafo para uma dada rotina.

entrada: blocos básicos
saída: Grafo de Fluxo

1. S = { }
2. Escolha o bloco básico que contenha como líder o ponto de entrada da rotina $B_i = B_0$.
3. Para todo B_j pertencente ao conjunto de blocos básicos defina uma aresta $B_i \rightarrow B_j$ se:

(a) existir um desvio da última instrução de B_i para a primeira instrução de B_j

(b) B_i preceder imediatamente B_j na ordem do programa

4. Acrescente B_i a S. Escolha um novo bloco básico B_i. Repita o passo 3 até que S contenha todos os blocos básicos.

5. Acrescente o bloco *entry* e a aresta entry $\rightarrow B_0$ ao grafo.

6. Acrescente o bloco *exit* ao grafo. Para cada bloco final B_i (bloco básico sem sucessor) acrescente a aresta $B_i \rightarrow$ *exit*.

Bloco Básico Estendido

Um bloco básico estendido é uma seqüência máxima de instruções começando com um líder onde na seqüência não contemos nós de junção (com exceção do líder). Um nó de junção é definido como um nó com grau de entrada maior que 2 - ou seja, possui mais de um ancestral.

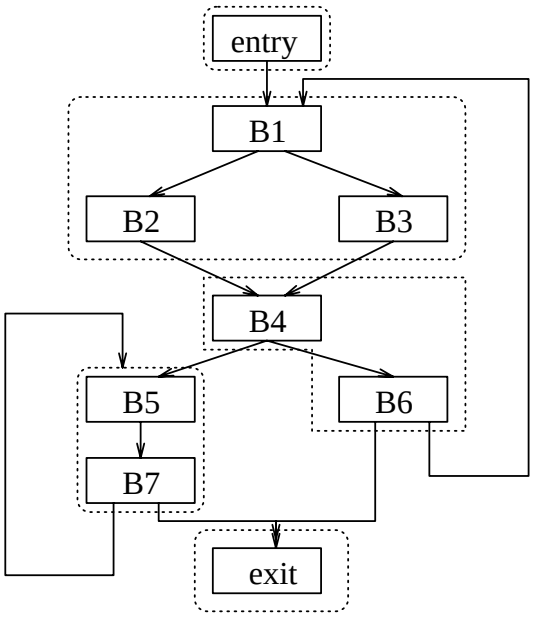


Figura 3: Grafo de Fluxo com blocos básicos estendidos.

Apresentamos a seguir um algoritmo para a determinação de blocos básicos estendidos.

```
EbbRoots := { r } : set of node
AllEbbs  := {   } : set of (node X set of node)

while EbbRoots <> { } do
```

```
        x := # EbbRoots : node
        EbbRoots -= { x }
        if for all S in AllEbbs (S@1 <> x) then
           AllEbbs U= {  < x, Buid_Ebb(x) >  }
        fi
od

procedure Build_Ebb(r : in node) returns set of node
begin
    Ebb = { } : set of node
    Add_Bbs(r, Ebb)
    return Ebb
end

procedure Add_Bbs(r : in node, Ebb : inout set of node)
begin
    x : node
    Ebb U= { r }
    for each x in Succ(r) do
        if | Pred(x) | = 1 & x in Ebb then
           Add_Bbs(x, Ebb)
        elif x not in EbbRoots then
           EbbRoots U= { x }
        fi
    od
end
```

3 Caminhamentos

Caminhar em um grafo é definir uma ordem em que os nós serão acessados. Nesta seção apresentaremos as formas mais comuns de caminhamento em grafos utilizadas pelos diversos algoritmos de análise de fluxo:

- busca em profundidade (depth first search)
- pré-ordem
- pós-ordem
- busca em amplitude (breadth first search)

Busca em Profundidade

A busca em profundidade permite o acesso aos descendentes de um nó do grafo antes de qualquer vizinho que não seja seu descendente. Dado, por exemplo, o grafo da Figura 4(a) a

Figura 4(b) representa uma possível busca em profundidade - o número associado a cada nó será denominado **número de profundidade** (depth-first number).

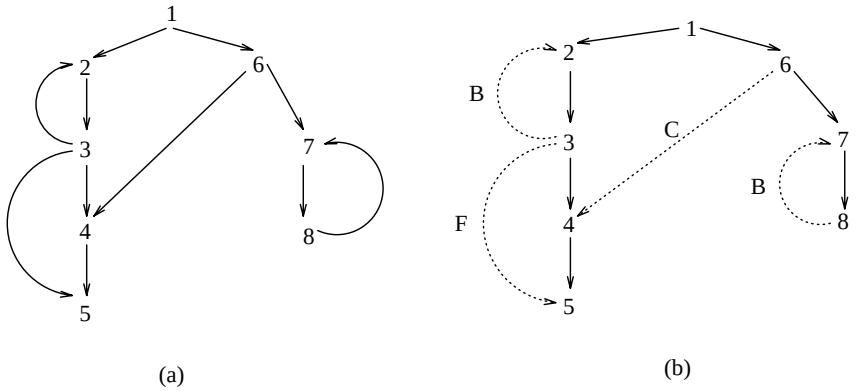


Figura 4: Busca em Profundidade.

Denominamos *Árvore Geradora* (Depth-First Spanning Tree) a árvore obtida por uma busca em profundidade. Note que esta árvore pode não ser única para um dado grafo. Por exemplo, o grafo da Figura 5(a) tem 2 representações diferentes como mostrado nas Figuras 5(b) e (c).

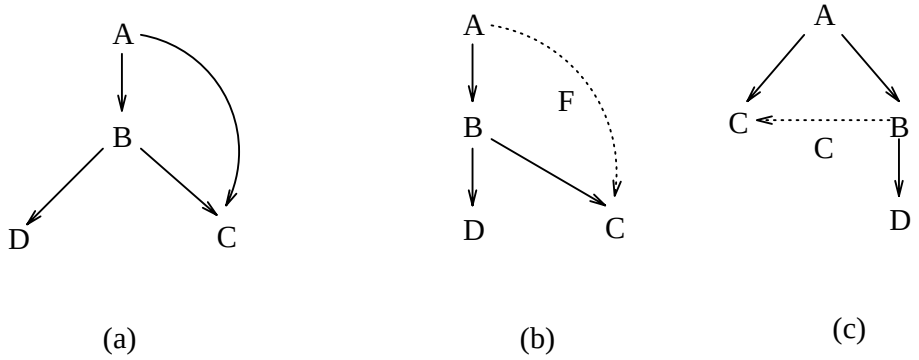


Figura 5: Árvores geradoras.

O algoritmo apresentado abaixo constrói a arvore geradora para um dado grafo G$< N, E >$.

```
Visit: node -> boolean

procedure Depth_First_Search(x : in node)
begin
   y : node
   Process(x)
```

```
    Visit(x) := true
    for each y in Succ(x) do
       if !Visit(y) then
          Depth_First_Search(y)
       fi
    od
end
```

Caminhamentos em Pré-Ordem e Pós-Ordem

O Caminhamento em pré-ordem é uma busca na qual cada nó é processado antes de seus descendentes. Como exemplo, podemos tomar a seqüência entry, B1, B2, B3, B4, B5, B6 e exit do grafo exibido na Figura 2. Analogamente, um caminhamento em pós-ordem é aquele em que cada nó é processado após seus descendentes. Novamente, da Figura 2, podemos obter a nova seqüência exit, B6, B5, B2, B4, B3, B1, entry.

Note que estas buscas não são únicas: entry, B1, B3, B2, B4, B6, B5, exit é um outro caminhamento em pré-ordem assim como exit, B5, B6, B4, B3, B2, B1, entry é um outro caminhamento em pós-ordem.

A rotina Depth-First-Search-PP() dado a seguir é uma instância da busca em profundidade que calcula tanto a árvore geradora como os caminhamentos em pré e pós-ordem de um grafo $G = < N, E >$. Ressaltamos que a cada aresta é associado um rótulo - que permite diferenciar as arestas pertencentes a árvore geradora das demais. Arestas pertencentes à árvore geradora são rotuladas por tree. As demais arestas são assim classificadas:

- forward edges: são rotuladas por **F** direcionando o nó para um descendente direto;
- back edges: são rotuladas por **B** direcionando o nó para seu ancestral;
- cross edges: são rotuladas por **C** conectando nós que não são nem ancestrais nem descendentes.

Note que estas aresta não existem na árvore geradora, mas pertencem ao grafo de fluxo original.

Algoritmo

```
i := 1  :  integer
j := 1  :  integer
Visit: node -> boolean
Etype: (node X node)  ->  enum { tree, forward, back, cross }

for each node in Gn : set of node
    Visit(node) := false
```

```
Procedure Depth_First_Search_PP(x : in node)
Begin
   y : node
   Visit(x) := true
   Pre(x) := j
   J += 1
   for each y in Succ(x) do
       if !Visit(y) then
          Depth_First_Search_PP( y )
          Etype(x -> y) := tree
       Elif  Pre(x) < Pre(y) then
          Etype(x -> y) := foward
       Elif  Post(y) = 0  then
          Etype(x -> y) := back
       Elif
          Etype(x -> y) := cross
       fi
   od
   Post(x) := i
   i += 1
end
```

Busca em Amplitude

Busca em amplitude é um caminhamento no qual todos os nós imediatamente descendentes são processados antes de qualquer um de seus descendentes. Tome, como exemplo, a ordem 1, 2, 6, 3, 4, 5, 7, 8 da Figura 4. O código abaixo apresentado define uma busca em amplitude através da rotulação numérica dos nodos de um grafo.

```
i := 2 :  integer
Procedure Breadth_First (s : in node)  returns  node -> integer
Begin
   t : node      W = { }  : set of node
   Order :  node  ->  integer
   Order(r) := 1
   J += 1
   for each t in Succ(s) do
       if Order(t) = nil then
          Order(t) := i
          i  +=  1
          W  U= { t }
       fi
   od
```

```
    for eacht in W do
        Breadth_First(t)
    od
    return Order
end
```

4 Dominadores

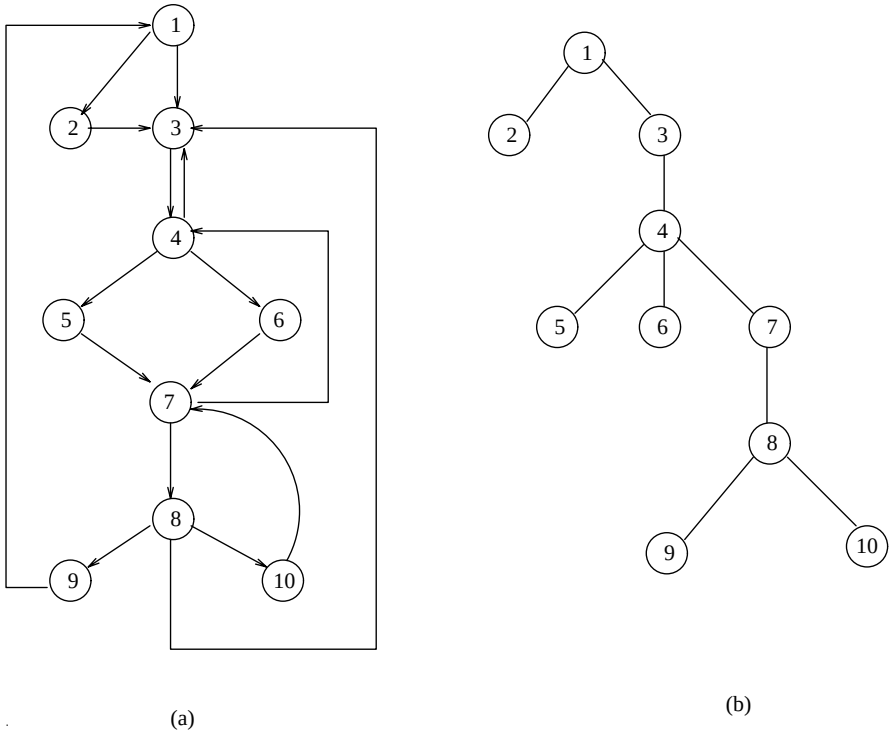


Figura 6: Árvore Dominadora.

Antes de considerarmos otimizações de loops, precisamos definir o que constitui um loop em um grafo de fluxo. Usaremos a noção de dominação para definir "loops naturais"e também classificar classes de grafo de fluxo, em particular as redutíveis.

Dizemos que um nó d domina i (d dom i) se todo caminho a partir de entry para i, incluir d. É fácil notar que a relação de dominação é reflexiva (todo nó domina a si mesmo), transitiva (se a dom b e b dom c, então a dom c), e antissimétrica (se a dom b, b não dom c necessariamente).

Um nó a é dito dominador imediato de b (a idom b) se e somente se a dom b e não existir um nó c tal que $c \neq a$ e $c \neq b$, **a dom c** e **c dom b**.

Outra relação importante é a de pós dominação: p pós domina i (p pdom i) se para todo caminho de i para exit incluir p, ou seja, i dom p. As informações de dominação podem ser

facilmente representadas por uma árvore denominada árvore dominadora (Figura 6). Segue abaixo um algoritmo para a determinação da árvore dominadora.

```
Considere G <N,E>

D( r ) = { r }
For n in N - { r } do D(n) := N

While changes to any D(n) occur do
      For n in N - { r } do
          D(n) = { r } U  intersection D(p),   p  in  Pred(n)
```

A execução do algoritmo produz, para a Figura 6(a), o seguinte resultado:

- D(1) = { 1 }
- D(2) = { 2 } $\bigcup$ { 1 } = { 1, 2 }
- D(3) = { 3 } $\bigcup$ ({ 1 } $\bigcap$ { 1, 2 } $\bigcap$ { 1, 2, 3 ... 10} $\bigcap$ { 1, 2, 3 .. 10 }) = { 1, 3 }

e assim sucessivamente.

5 Loops

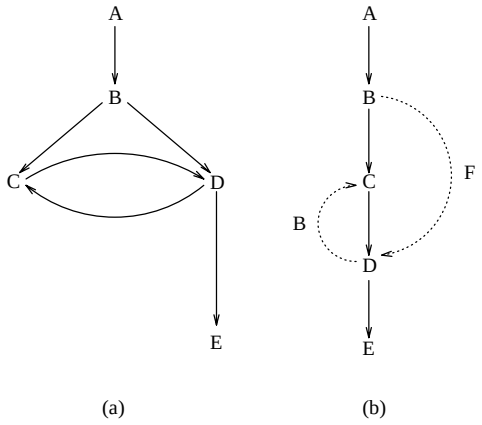


Figura 7: Loops.

Uma aplicação importante das relações de dominação está na determinação de loops para efeitos de otimização. A obtenção e extração de loops em um grafo de fluxo deve ser feita cudadosamente.

Note que o grafo da Figura 7(b) tem uma aresta de retorno (back edge) $d \rightarrow c$ tal que c não domina d. Embora esta aresta de retorno constitua um loop, este loop tem 2 pontos de entrada (c e d), logo não é um "loop natural". Podemos definir um loop natural como aquele que possui:

- um único nó de entrada denominado *header* - este nó domina todos os nós do loop e,
- possui pelo menos uma forma de iteração no loop, i.e., pelo menos uma aresta de retorno para o header.

Isto significa que uma forma de se encontrar todos os loops em um grafo é procurar por arestas de retorno onde, dado $a \rightarrow b$ a aresta de retorno, b dom a.

Dado uma aresta de retorno $n \rightarrow d$ definimos o loop natural como a aresta d mais o conjunto de nós que podem alcançar n sem passar por d (d o header do loop). Mostramos a seguir um algoritmo para a determinação de um loop natural para uma aresta de retorno $n \rightarrow d$.

```
procedure insert(m)
begin
   if m is not in loop then
      loop := loop U { m }
      push m onto stack
   fi
end

stack := empty
loop := { d }
insert(n)
while stack is not empty do
   pop m, the first element of stack, off stack
   for each predecessor p of m do insert(p)
od
```

Loops Aninhados

É fácil verificar que, se dois loops naturais não tem o mesmo header, ou eles são disjuntos ou então um está inteiramente contido no outro - o que nos leva a noção natural de loops aninhados.

Um problema pode ocorrer quando dois loops tem o mesmo header. Neste caso é difícil avaliar a relação existente entre os mesmos analisando apenas o grafo. Considere os loops mostrados na Figura 8. Este grafo poderia corresponder ao seguinte código:

```
B1:  if (i < j)
        goto B2
     else if (i > j)
        goto B3
     else goto B4
B2:  . . .
     i++;
```

```
     goto B1;
B3:  . . .
     i--;
     goto B1;
B4:  . . .
```

Note que no programa temos apenas um loop. Posteriormente será mostrado como efetuar uma análise mais detalhada deste problema.

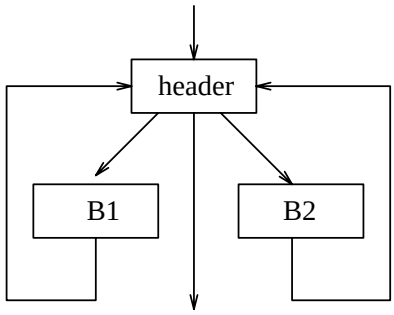


Figura 8: Dois loops naturais com o mesmo header.

PreHeader

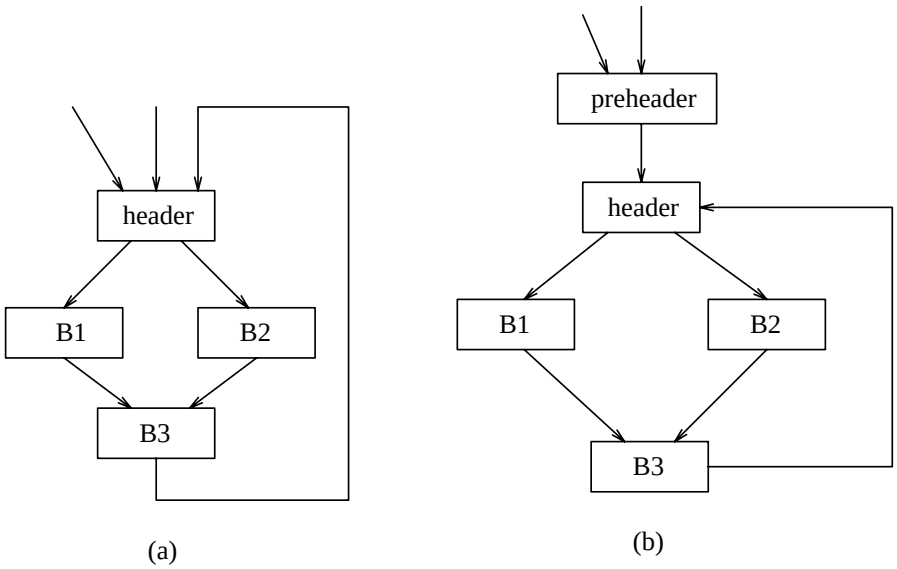


Figura 9: Loop (a) sem e (b) com preheader.

Várias otimizações exigem a movimentação de instruções de uma seqência para fora do loop

imediatamente antes do seu header. Para garantir que tais movimentações possam sempre ser efetuadas introduziremos o conceito de preheader.

Um preheader é um novo bloco (inicialmente vazio) que é conectado ao header do loop de tal forma que todas as arestas externas ao loop incidentes no header são direcionadas ao preheader. Uma nova aresta (única) conecta o preheader ao antigo header. A Figura 9 mostra um exemplo de um loop antes e após a inserção de um preheader.

1 Introdução a Análise de Fluxo de Controle

Análise de Fluxo de Controle

Flávia Peligrinelli Ribeiro

Análise de Fluxo de Controle

Flávia Peligrinelli Ribeiro
Departmento de Ciência da Computação
Universidade Federal de Minas Gerais

07 de junho de 1999

1 Introdução

A otimização de código requer que o compilador tenha componentes que possam construir um entendimento global de como o programa usa os recursos. O compilador deve caracterizar o fluxo de controle de programas e as manipulações que eles fazem sobre os seus dados, de forma que qualquer generalidade não usada que poderia resultar em uma compilação não otimizada pode ser retirada. Mecanismos menos eficientes porém mais genéricos são substituídos por mecanismos mais eficientes e específicos.

Há duas formas principais de se fazer análise de fluxo de dados em rotinas simples. Elas usam o conceito de bloco básico que compõe a rotina e então a construção de seu grafo de fluxo. A primeira abordagem usa dominadores para descobrir *loops* e os usa na otimização. A segunda abordagem, chamada análise de intervalos[1], inclui uma série de métodos que analizam toda a estrutura da rotina e que a decompõe em regiões chamadas intervalos. Esses intervalos formam uma árvore, chamada árvore de controle, que é útil e torna a análise de dados mais rápida. A análise de intervalo mais sofisticada, chamada análise estrutural[1], classifica basicamente todas estruturas de fluxo de controle de uma rotina.

A maior parte dos compiladores otimizadores atuais usam dominadores e análise de fluxo de dados iterativo.

Na Seção 2, vamos falar sobre análise de intervalos e na Seção 3 abordaremos a análise estrutural. E por fim faremos uma conclusão comentando resumindamente essas análises.

2 Análise de Intervalo e Árvores de Controle

A análise de intervalo é feita durante a análise de fluxo de controle. Dependendo da abordagem adotada, ela divide o grafo de fluxo em regiões de vários tipos (dependendo de cada abordagem), transformando cada região em um novo nodo, geralmente chamado de nodo abstrato[1] e, finalmente, trocando as arestas de entrada ou saída da região por arestas de entrada ou saída correspondentes ao nodo abstrato.

O grafo de fluxo resultante de uma ou mais transformações desse tipo é chamado de grafo de fluxo abstrato. As transformações são aplicadas somente a um grafo de fluxo de cada vez, ou a subgrafos disjuntos em paralelo.

O resultado da aplicação de uma seqüência de transformações produz uma árvore de controle. Uma árvore de controle é definida da seguinte forma:

[1] O nodo é chamado de abstrato porque ele abstrai a estrutura interna da região que representa

1. a raiz é um grafo abstrato representando o grafo de fluxo original;
2. as folhas são blocos básicos individuais;
3. os nodos entre a raiz e as folhas são nodos abstratos que representam as regiões do grafo de fluxo;
4. as arestas representam o relacionamento entre o nodo abstrato e as regiões que são seus descendentes, ou seja, que foram abstraídas para formá-lo.

Uma das formas mais simples de análise de intervalo é conhecida como T1-T2. Ela é composta por duas transformações básicas:

1. T1: transforma um nodo *self-loop* em um único nodo. (Veja Figura 1.)

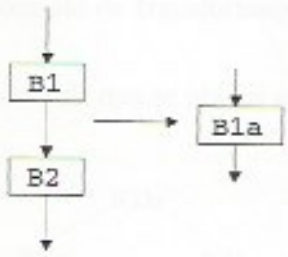


Figure 1: Transformação T1

2. T2: transforma uma seqüência de dois nodos tal que o primeiro é o único predecessor do segundo em um único nodo. (Veja Figura 2.)

Figure 2: Transformação T2

A Figura 3 mostra um exemplo de como essas transformações funcionam no grafo de fluxo:

- Os blocos B1 e B2 sofreram uma transformação T2 para o bloco B1a e os blocos B3 e B4 sofreram uma transformação T2 para o bloco B3a.
- O bloco B3a sofreu uma transformação T1 para o bloco B3b.
- Os blocos B1a e B3b sofreram uma transformação T2 para o bloco B1b.

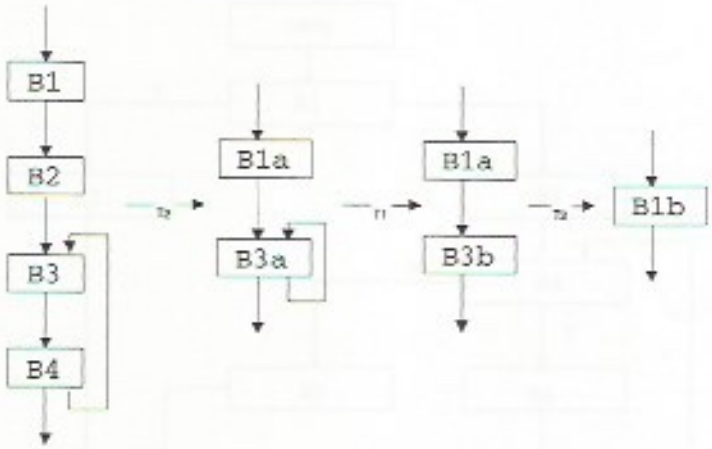


Figure 3: Exemplo de transformações T1 e T2

A Figura 4 mostra a árvore de controle que se obtém a partir das transformações do grafo de fluxo.

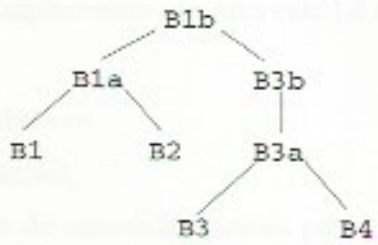


Figure 4: Árvore de controle T1-T2 para o grafo de fluxo da Figura 3

2.1 Intervalo Máximo e Intervalo Mínimo

A análise de intervalo explicada anteriormente usou na sua análise intervalos máximos e ignorou a existência dos conceitos de irredutibilidade e de regiões impróprias [2].

Um intervalo máximo $IM(h)$ é máximo, possui uma única entrada e um único bloco de entrada h e todos os caminhos fechados contém h. Dessa forma, $IM(h)$ é um *loop* natural que possui como entrada do nodo h além de estruturas *dangling* acíclicas.

A Figura 5 mostra um grafo de fluxo. O intervalo máximo nesse grafo $IM(B4) = B4, B6, B5, exit$.

O nodo $B6$ está incluído pois o único caminho fechado contendo $B4$ é aquele que consiste de $B4 \rightarrow B6$ e $B6 \rightarrow B4$. B5 e exit estão incluídos porque caso contrário o subgrafo não seria máximo.

[2] Alguns padrões de fluxos de controle produzem grafos de fluxos que são irredutíveis. Tais padrões são chamados de regiões impróprias, e, em geral, são regiões do grafo de fluxo que possuem entradas múltiplas com componentes fortemente conectados.

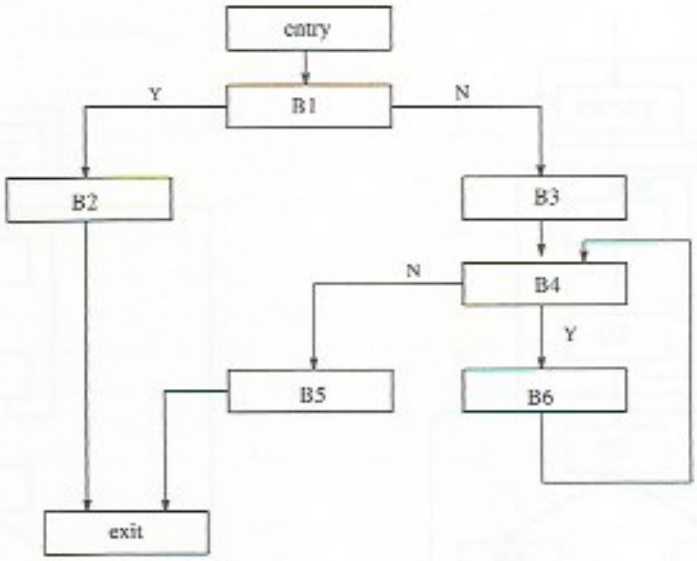


Figure 5: Um exemplo

Um intervalo mínimo, ou simplesmente um intervalo I é definido como sendo:

1. um *natural loop*,
2. um subgrafo acíclico máximo ou
3. uma região mínima irredutível.

Um intervalo mínimo difere do intervalo máximo pois o último inclui os sucessores dos nodos do *loop* que não estão propriamente no *loop* e que também não são líderes (*headers*) de intervalos máximos enquanto que o intervalo mínimo os exclui. As Figuras 6 e 7 mostram para um mesmo grafo de fluxo os intervalos máximos e mínimos.

2.2 Algoritmo

Os passos básicos para se fazer uma análise de intervalo são:

1. Aplique o algoritmo *postorder traversal* ao conjunto de nodos do grafo de fluxo, procurando os líderes dos *loops* a cada nodo único e os líderes das regiões impróprias a cada conjunto de mais de um nodo.
2. Para cada cada líder de *loop* encontrado, construa seu *loop* natural e reduza-o a uma região abstrata do tipo *loop* natural.
3. Para cada conjunto de entradas de uma região imprópria, construa o componente mínimo fortemente conectado do grafo de fluxo contendo todas as entradas e reduza-o a uma região abstrata do tipo região imprópria.
4. Para o nodo de entrada e para cada descendente imediato de um nodo na forma *loop* natural ou região irredutível, construa o grafo acíclico máximo com o nodo de entrada como raiz. Se o grafo resultante possuir mais de um nodo reduza-o a uma região abstrata do tipo região acíclica.

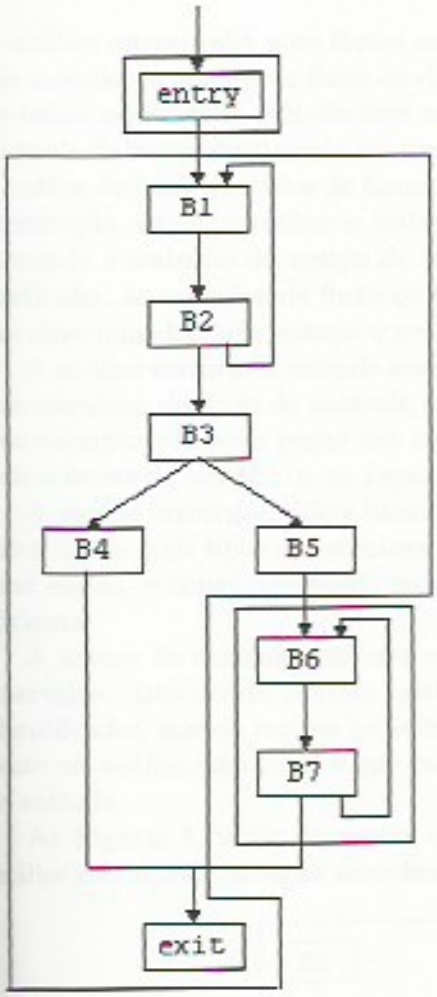


Figure 6: Um exemplo de Intervalos Máximos

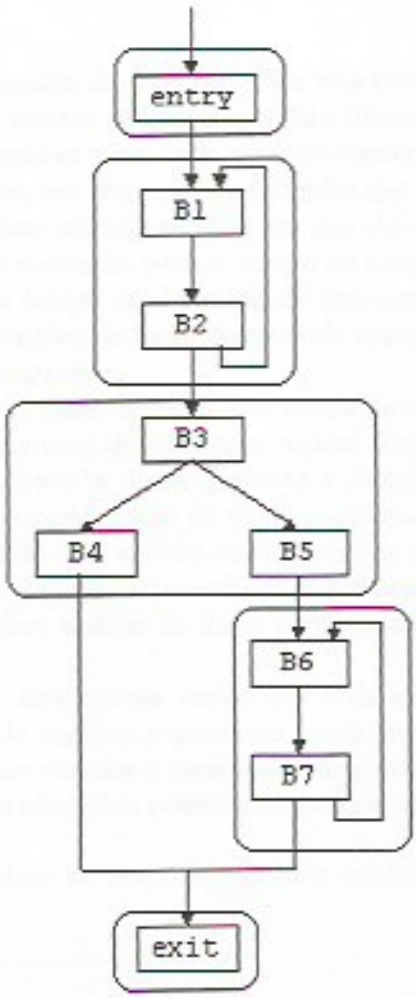


Figure 7: Um exemplo de Intervalos Mínimos

5. Repita esse processo até que termine.

O término do algoritmo é garantido pois ou o grafo de fluxo é acíclico ou ele contém um tipo de loop:

- se é acíclico, o processo termina com a iteração corrente,
- se possui um ou mais ciclos, pelo menos uma redução irá ocorrer para um *loop* natural ou para uma região imprópria durante cada iteração, reduzindo o número de ciclos de pelo menos um. Cada grafo de fluxo contém somente um número finito de ciclos.

3 Análise Estrutural

A análise estrutural é uma forma mais refinada de análise de intervalo. Seu objetivo é fazer um método de análise de fluxo de dados [3] dirigido a sintaxe aplicável a código intermediário de baixo nível. Esse método tem a vantagem de fornecer para cada tipo de construção de controle de fluxo estruturado em uma linguagem fonte, um conjunto de fórmulas que executa a análise de fluxo de dados de forma mais eficiente. Esse método estende um dos objetivos da otimização, isto é, transfere o trabalho do tempo de execução para o tempo de compilação, movendo o trabalho do tempo de compilação para o tempo de definição de linguagem. Em particular, as equações de fluxo de dados para construções de fluxo de controle estruturadas são determinadas pela sintaxe e pela semântica da linguagem.

A análise estrutural estende essa abordagem para grafos de fluxo arbitrários descobrindo sua estrutura de fluxo de controle e provendo uma maneira de lidar com regiões impróprias. Por exemplo, ele pode pegar um *loop* feito de `if's`, `goto's` e `assignments` e descobrir que tem a forma de um `while` ou `repeat`, mesmo que sua sintaxe não dê nenhuma pista disso.

A análise estrutural difere basicamente da análise de intervalo no sentido em que ela identifica muito mais tipos de estrutura de controle além de *loops* formando para cada estrutura, uma região, e como resultado, provê a base para fazer análise de fluxo de dados de forma eficiente.

A árvore de controle que essa análise constrói é tipicamente maior que a da análise de intervalos. Isto ocorre porque existem mais tipos de regiões, e portanto, mais regiões são identificadas, mas as regiões individualmente são mais simples e menores. Um ponto importante na análise estrutural é que cada região que ela identifica possui exatamente um ponto de entrada.

As Figuras 8, 9 são exemplos de estruturas típicas de controle cíclico e acíclico que a análise estrutural consegue reconhecer.

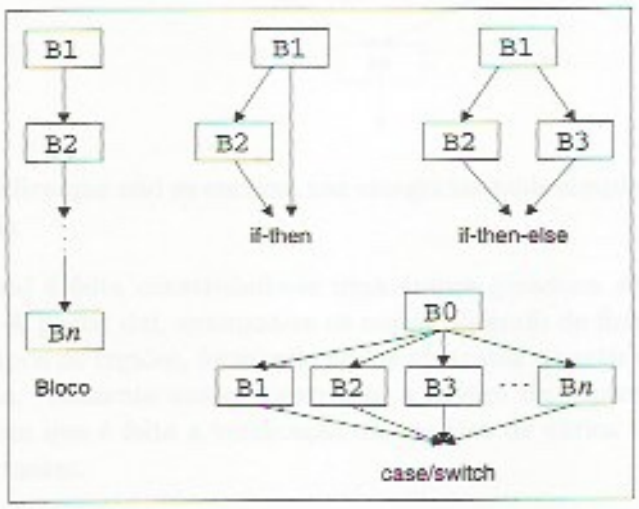


Figure 8: Tipos de regiões acíclicas usadas na análise estrutural

Um outro tipo de intervalo usado na análise estrutural é o intervalo próprio, uma estrutura acíclica arbitrária, por exemplo, um que no contenha ciclos e possa ser reduzido para qualquer

[3] desenvolvido por Rosen[2] para ser usado em árvores sintáticas.

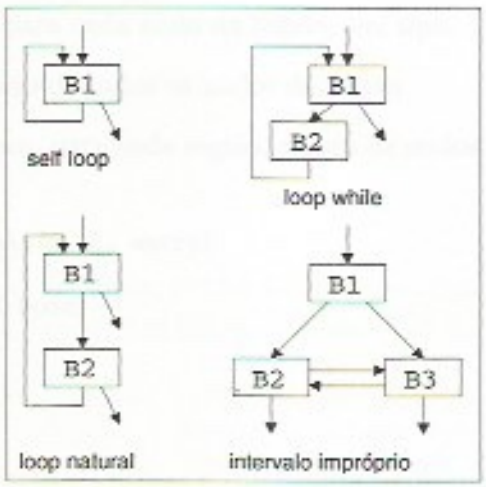


Figure 9: Tipos de regiões acíclicas usadas na análise estrutural

um dos casos simples acíclicos. (Veja a Figura 10.)

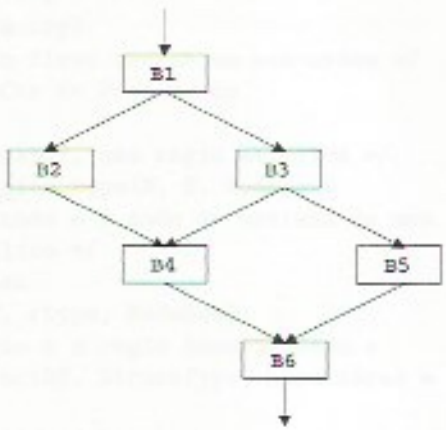


Figure 10: Região acíclica que não se encaixa nas categorias mais simples, então é identificada como região imprópria

A análise estrutural é feita construindo-se uma árvore geradora *depth first* para o grafo de fluxo em questão. A partir daí, examina-se os nodos do grafo de fluxo em pós-ordem para instâncias de vários tipos de regiões, formando nodos abstratos a partir deles e transformando as arestas de conexão. Somente então constrói-se a árvore de controle correspondente ao processo. A ordem com que é feita a verificação das regiões de vários tipos e a maneira com que é feita são importantes.

3.1 O Algoritmo de Sharir

Primeiramente, são construídas quatro estruturas de dados enquanto o grafo de fluxo é analisado.

- `StructureOf`: fornece, para cada nodo, o nodo abstrato da região que imediatamente o contém.

- `StructTypes`: fornece, para cada nodo da região, seu tipo.
- `Structures`: é o conjunto de todos os nodos da região
- `StructureNodes`: fornece, para cada região, a lista de nodos contidos nela.

```
procedure Structural_Analysis(N, E, entry)
   N: in the set of Node
   E: in the set of (Node X Node)
   entry: in Node

begin
   m, n, p: Node
   rtype: RegionType
   NodeSet, ReachUnder: set of Node
   StructOf:= StructType:= Structures:= StructNodes:= 0
   CTNodes:= N; CTEdges:= 0
   repeat
      Post:= 0; Visit:= 0
      PostMax:= 0; PostCtr:= 1
      DFS_postorder(N, E, entry)
      /* caminhamento depth first search em pos-ordem */
      while |N| > 1 & postCtr <= Postmax do
        n:= post(PostCtr)
        /* localiza, se existir, uma regio aciclica */
        rtype:= Acyclic_region_type(N, E, NodeSet)
        /* determina se o nodo e o nodo de entrada de uma
 estrutura de controle aciclico */
        if rtype != nil then
           p:= Reduce(N, E, rtype, NodeSet)
           /* cria uma regio n a regio identificada e
 atualiza as estruturas StructOf, StructType, Structures e
 StructNodes */
           if entry pertence nodeSet then
              entry:= p
           fi
        else
        /* localiza, se existir, uma regio aciclica */
           ReachUnder:= {n}
           for each m pertencente N do
              if StructOf(m) = nil & Path_Back(m,n) then
                 ReachUnder u={m}
              fi
           od
           rtype:= Ciclic_Region_Type(N, E, ReachUnder)
           /* determina se o nodo e o nodo de entrada de
 uma estrutura de controle ciclico */
           if rtype != nil then
              p:= Reduce(N, E, rtype, NodeSet)
              if entry pertence nodeSet then
                 entry:= p
              fi
```

```
            else
               PostCtr += 1
            fi
         fi
      od
   until |N| = 1
end
```

O programa acima assume que serão usadas os tipos de regiões mostradas nas Figuras 8 e 9. Outras regiões podem ser usadas quando apropriadas à linguagem que está sendo processada.

O algoritmo primeiramente inicializa a estrutura de dados descrita anteriormente, que guarda a estrutura hierárquica do grafo de fluxo e as estruturas que representam a árvore de controle (`CTNodes e CTEdges`). Ele então faz uma busca *depth first* no grafo de fluxo para construir a *postorder traversal* dos nodos do grafo de fluxo. Depois, em várias passadas pelo grafo de fluxo, ele identifica cada região e o transforma em um único nodo abstrato de região. Se uma redução é feita, ele conserta os conjuntos de nodos e arestas no grafo de fluxo e se necessário, o *postorder traversal* e processa o grafo novamete. O algoritmo repõe as arestas de entrada de uma região por arestas do novo nodo abstrato que o substituiu e arestas de saída por aresta do novo nodo. Em paralelo, ele constroi a árvore de controle.

Como um exemplo de análise estrutural, considere o grafo de fluxo da figura 11, 12.

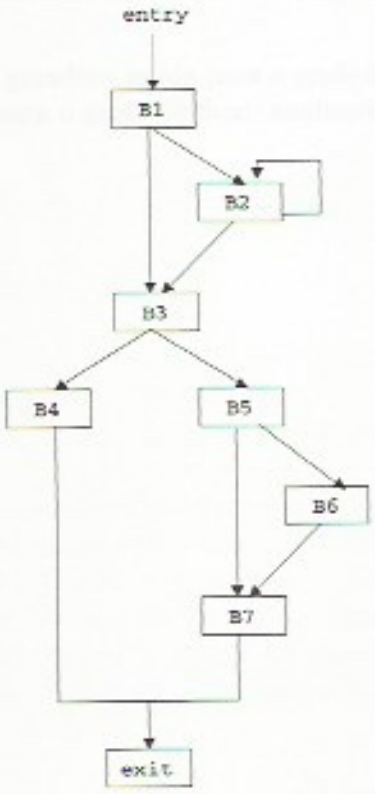


Figure 11: Análise estrutural de um grafo de fluxo

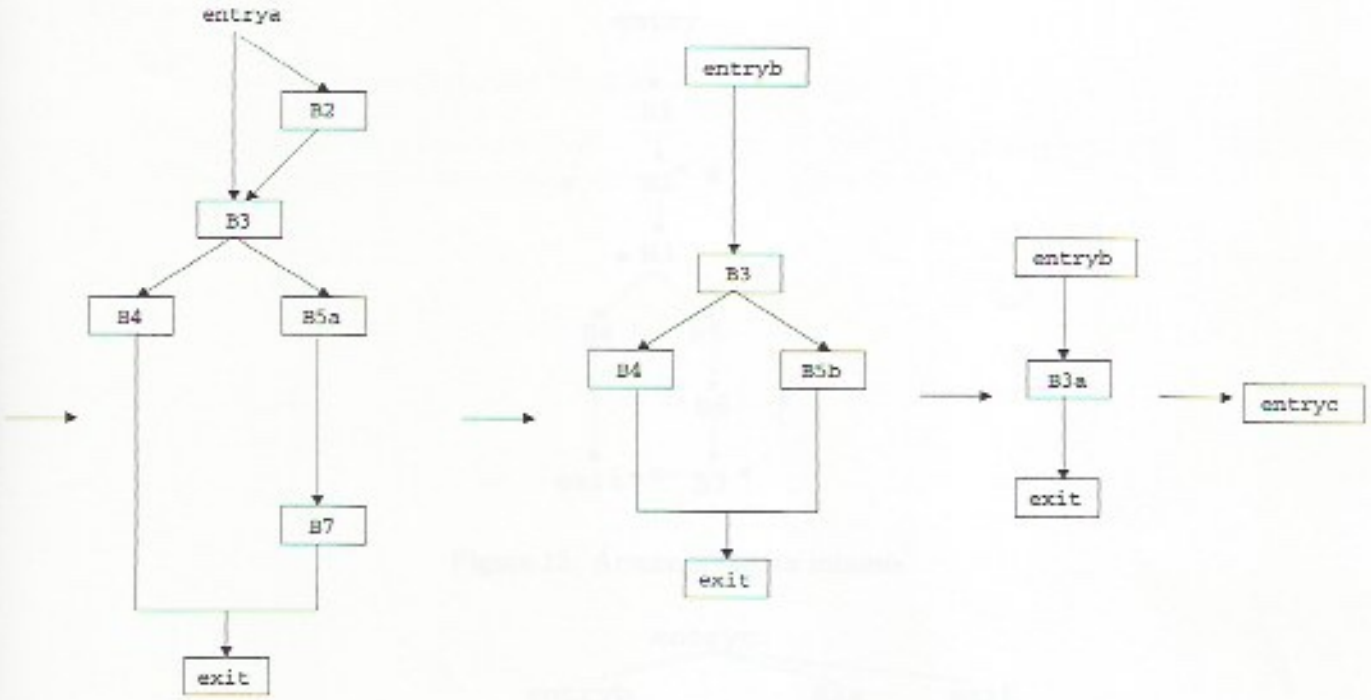


Figure 12: Análise estrutural de um grafo de fluxo

A Figura 13 mostra a árvore geradora mínia para o grafo de fluxo da Figura 11 e a Figura 14 mostra a árvore de controle para o grafo de fluxo analisado.

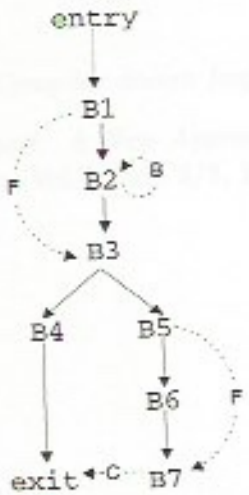


Figure 13: Árvore geradora mínima

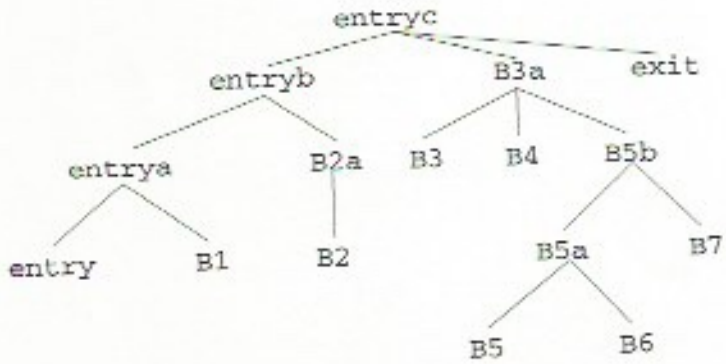


Figure 14: Árvore de controle

4 Conclusão

A análise de controle é o primeiro passo com o que se refere a otimização de código.

A otimização tem como requisito a habilidade de caracterizar o fluxo de controle de programas e as manipulações feitas nos seus dados, tal que generalidades não usadas possam ser removidas e substituídas por comandos mais rápidos.

Nesse artigo, nós mostramos a análise de intervalos, que inclui uma série de métodos que analisam toda a estrutura da rotina e a decompõe em regiões chamadas intervalos. Essa estrutura de intervalos aninhados formam a árvore de controle. A análise de intervalo mais sofisticada é chamada de análise estrutural e essencialmente ela classifica todas as estruturas de fluxo de controle em uma rotina.

References

[1] Muchnick, Steven S. *Advanced Compiler design Implementation*

[2] Sharir, Micha. *Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers*, Computer Languages, Vol.5, Nos. 3/4, 1980, pp141-153

2 Análise de Fluxo de Controle

Análise de Fluxo de Dados

Fábio Tirelo

Análise de Fluxo de Dados*

Fabio Tirelo

7 de junho de 1999

Resumo

A *análise de fluxo de dados* tem por objetivo fornecer ao otimizador informações sobre como o programa manipula os seus dados. A partir destas informações, o otimizador pode fazer transformações sobre o programa, preservando a sua semântica, de modo a gerar um código mais eficiente em termos de espaço e tempo de execução. Neste trabalho, apresentaremos os principais conceitos e técnicas de análise de fluxo de dados, assim como algumas de suas aplicações.

1 Introdução

O propósito da *análise de fluxo de dados* é prover informações sobre como um procedimento, ou um grande segmento de programa, manipula seus dados. Por exemplo, a análise de propagação de constantes tenta determinar se todas as atribuições a uma variável resultam no mesmo valor constante. Se for, o uso desta variável poderia ser substituído pela constante.

O espectro das análises de fluxo de dados possíveis variam desde a execução abstrata de um procedimento, que poderia determinar, por exemplo, que ele computa a função fatorial, até análises muito mais simples, como o problema de definições que podem alcançar algum uso (*reaching definitions*).

Em todos os casos, devemos ter certeza que a análise de fluxo de dados nos fornece informações que representam corretamente o que o procedimento faz, isto é, ela não deve nos dizer que uma transformação de código é segura, quando, na verdade, não é. Isto é garantido por meio de um projeto cuidadoso das equações de fluxo de dados, sabendo que, se suas soluções não são uma representação exata, são pelo menos aproximações *conservativas* da manipulação de dados do procedimento.

Entretanto, para obtermos o maior lucro possível da otimização, desejamos que um problema de fluxo de dados seja conservativo e, ao mesmo tempo, o mais agressivo possível. Portanto, sempre tentaremos ser o mais agressivo possível nas informações que computamos, para obtermos o maior benefício possível a partir das análises e das transformações para melhoria do código, sem deixarmos de ser conservativos, para não transformar código correto em código incorreto.

Este texto está dividido da seguinte forma:

- na Seção 2, mostraremos a taxonomia de problemas de fluxo de dados, assim como apresentaremos alguns dos problemas mais importantes para otimização;
- na Seção 3, falaremos sobre a análise estrutural, que é um método importante de modelagem dos problemas de fluxo de dados;
- a Seção 4 aborda a análise de intervalos, que é uma forma mais simples de análise estrutural;
- a Seção 5 trata da abordagem de análise *slotwise*, que aumenta a eficiência de tempo e de espaço na resolução de alguns problemas de análise de fluxo de dados;

*Seminário apresentado na disciplina *Tópicos em Compiladores: Otimização de Código*

- na Seção 6, mostramos técnicas para lidar com estruturas de dados complexas durante a análise de fluxo de dados;
- na Seção 7, citaremos alguns tipos de análises mais ambiciosas;
- e, finalmente, na Seção 8, mostraremos algumas otimizações que podem ser feitas a partir das informações obtidas na análise de fluxo de dados.

As Seções 2 a 7 foram retiradas de [3] e a Seção 8 foi retirada de [1].

2 Taxonomia de Problemas de Fluxo de Dados e Métodos de Solução

Problemas de análise de fluxo de dados podem ser classificados de diversas formas, incluindo as seguintes:

1. as informações que desejamos obter;
2. se eles são relacionais ou envolvem atributos independentes;
3. os tipos de reticulados[1] usados e os significados associados aos elementos do reticulado e as funções definidas sobre estes elementos;
4. a direção do fluxo de informações: na direção da execução do programa, problemas diretos, na direção contrária à execução, problemas inversos, ou nas duas direções, problemas bidirecionais.

Quase todos os problemas que consideramos são do tipo *atributos independentes*, isto é, eles associam um elemento do reticulado a cada objeto de interesse, seja uma definição de variável, uma computação de expressão, etc. Apenas alguns poucos problemas requerem que o estado de fluxo de dados de um procedimento em cada ponto seja expresso por uma relação que descreve as relações entre os valores das variáveis, ou algo semelhante. Os problemas relacionais têm complexidade computacional muito maior que os problemas independentes de atributos.

Da mesma forma, quase todos os problemas que nós consideramos são unidirecionais, ou diretos ou inversos. Problemas bidirecionais requerem propagação para frente e para trás ao mesmo tempo e são significativamente mais complicados de formular, entender e resolver. Felizmente, em otimização, problemas bidirecionais são raros. A instância mais importante é a formulação clássica da eliminação da redundância parcial, que já foi ultrapassada por versões mais modernas, que utilizam somente análise unidirecional.

Entre as análises de fluxo de dados mais importantes para otimização de programas, estão as descritas abaixo.

Reaching Definitions – determina quais definições de variáveis, atribuições à variável, podem alcançar algum uso da variável no procedimento. Este é um problema direto que utiliza um reticulado de vetores de *bits*, onde cada *bit* corresponde a uma definição de variável.

Expressões Disponíveis – determina quais expressões estão disponíveis em cada ponto de um procedimento, de modo que, em todos os caminhos a partir da entrada até o ponto, há uma avaliação da expressão e nenhuma das variáveis que ocorrem na expressão foram atualizadas após a última avaliação da expressão. Expressões disponíveis é um problema que usa um reticulado de vetores de *bits* em que um *bit* é associado a cada definição de uma expressão.

[1] Reticulados são as estruturas algébricas que formam a base matemática da análise de fluxo de dados. Estas estruturas possuem certas propriedades que garantem, por exemplo, que os algoritmos de análise de fluxo de dados terminam.

Variáveis Vivas – determina, para uma dada variável e um dado ponto no programa, se há algum uso da variável em algum caminho a partir do ponto até a saída. Este é um problema inverso que utiliza vetores de *bits* em que cada uso da variável é associado a uma posição.

Usos Expostos – determina quais usos de variáveis em pontos particulares são alcançados por definições particulares. É um problema inverso que usa vetores de *bits* com um *bit* correspondendo a cada uso da variável. É o dual do problema *reaching definitions*, pois um liga definições a usos e o outro usos a definições.

Análise de Propagação de Cópias – determina se, em todo caminho a partir de uma atribuição de cópia da forma $x \leftarrow y$ até um uso da variável x, não há atribuições a y. Este é um problema direto, que usa vetores de *bits* em que cada *bit* representa uma cópia.

Análise de Propagação de Constantes – determina se em todo caminho a partir de uma atribuição de uma constante a uma variável da forma $x \leftarrow const$ até um uso de x, todas as atribuições a x lhe atribuem o valor *const*. Este é um problema de fluxo direto.

Análise de Redundância Parcial – determina se computações são realizadas duas ou mais vezes em algum caminho de execução sem que os operandos sejam modificados entre as computações. Este é um problema de fluxo bidirecional que utiliza vetores de *bits*, em que cada posição representa uma computação de expressão.

Neste texto, iremos nos concentrar no método de análise estrutural para problemas de fluxo de dados, que permite resolver todos os problemas acima.

3 Análise Estrutural

A *análise estrutural* utiliza as informações detalhadas sobre as estruturas de controle, desenvolvida pela análise de fluxo de controle estrutural, para produzir equações que representam os efeitos do fluxo de dados das estruturas de controle. A solução destas equações nos fornece as informações que desejamos.

Mostraremos aqui a análise direta, que é a mais simples. A análise inversa é um pouco mais complicada, pois as construções de fluxo de controle têm sempre uma única entrada, mas não necessariamente uma única saída. Contudo, a obtenção das equações é semelhante à direta, o que justifica a não inclusão neste texto.

3.1 Análise Estrutural: Análise Direta

Inicialmente, assumiremos que estamos realizando análise direta e que o efeito da combinação das informações de fluxo de dados, onde vários caminhos de fluxo de controle se encontram, é modelado pela operação *meet*, $\sqcap$. Esta operação é feita sobre elementos de *reticulados*. Por exemplo, para um reticulado de vetores de n bits, $\mathbf{BV}^n$, se $x, y \in \mathbf{BV}^n$, então $x \sqcap y$ é dada pela conjunção lógica (*and*) *bit* a *bit* entre x e y. Para um reticulado de funções de fluxos, $\mathbf{L}^F$, temos que se $f, g \in \mathbf{L}^F$, então, para todo $x \in \mathbf{L}$, $(f \sqcap g)(x) = f(x) \sqcap g(x)$.

A análise estrutural opera em duas fases. Na primeira fase, que denominamos fase *bottom-up*, determinamos as funções de fluxo de cada construção, a partir dos seus constituintes. Na segunda fase, denominada fase *top-down*, determinamos a entrada de cada bloco, a partir das construções envolventes.

Ao realizarmos a análise de fluxo de dados estrutural, a maioria dos grafos de fluxo que encontramos no primeiro passo são regiões simples como as mostradas na Figura 1. A *função de fluxo* F_B para um bloco básico representa a transformação das informações de fluxo de dados correspondentes à execução do bloco B.

Considere a construção `if-then` mostrada na Figura 2(a), com as funções de fluxo para cada construção dadas nos seus arcos de saída. A função de fluxo $F_{\texttt{if-then}}$, construída no primeiro

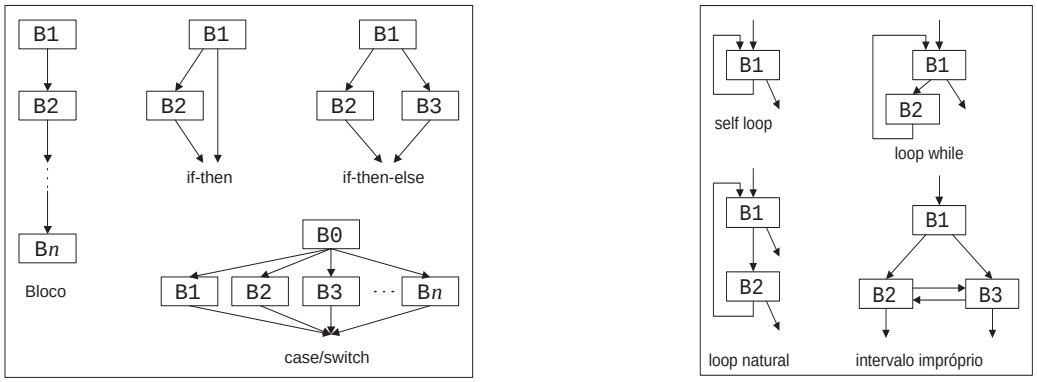


Figura 1: Alguns Tipos de Regiões Utilizadas em Análise Estrutural: (a) Regiões Acíclicas e (b) Regiões Cíclicas.

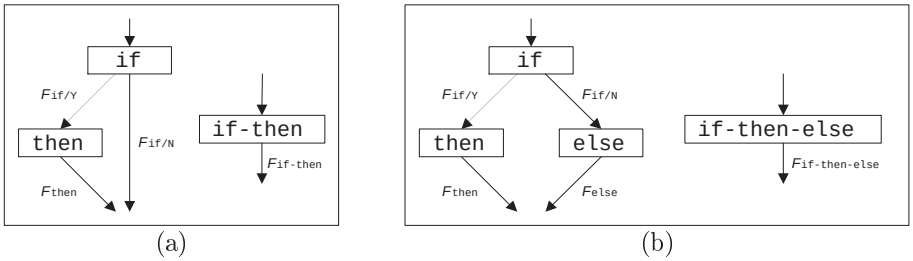


Figura 2: Funções de Fluxo para Análise Estrutural das Construções: (a) `if-then` e (b) `if-then-else`.

passo, está relacionada com as funções de fluxo dos componentes do `if-then` da seguinte forma:

$$F_{\texttt{if-then}} = (F_{\texttt{then}} \circ F_{\texttt{if/Y}}) \sqcap F_{\texttt{if/N}}$$

isto é, o efeito da execução da construção `if-then` é o efeito da execução da parte `if`, saindo pelo `Y`, seguido da execução da parte `then`, combinado com o efeito da execução da parte do `if` e saindo pelo `N`.

Note que podemos escolher entre distinguir ou não as saídas `Y` e `N` do `if` e ter funções de fluxo distintas $F_{\texttt{if/Y}}$ e $F_{\texttt{if/N}}$ para cada uma. Se escolhêssemos não distingui-las, deveríamos simplesmente ter uma única função de fluxo para o bloco `if`, $F_{\texttt{if}}$, ao invés de $F_{\texttt{if/Y}}$ e $F_{\texttt{if/N}}$. Neste caso, teríamos:

$$F_{\texttt{if-then}} = (F_{\texttt{then}} \circ F_{\texttt{if}}) \sqcap F_{\texttt{if}} = (F_{\texttt{if}} \sqcap id) \circ F_{\texttt{if}}$$

As duas abordagens são válidas. A primeira pode produzir informações mais precisas que a segunda, nos casos em que os desvios do `if` determinam os valores de fluxo de dados de nosso interesse como, por exemplo, na propagação de constantes ou na análise de verificação de limites. Nos exemplos abaixo, entretanto, utilizaremos a segunda abordagem.

As equações de fluxo de dados construídas no segundo passo nos diz, dadas as informações de fluxo de dados que entram na construção `if-then`, como propagá-las para a entrada de cada uma das suas substruturas. Elas são relativamente transparentes:

$$\begin{aligned} in(\texttt{if}) &= in(\texttt{if-then}) \\ in(\texttt{then}) &= F_{\texttt{if/Y}}(in(\texttt{if})) \end{aligned}$$

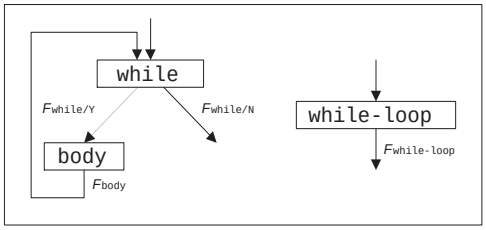


Figura 3: Funções de Fluxo para Análise Estrutural de um Loop `while`

ou se escolhêssemos não distinguir as saídas:

$$\begin{aligned} in(\texttt{if}) &= in(\texttt{if-then}) \\ in(\texttt{then}) &= F_{\texttt{if}}(in(\texttt{if})) \end{aligned}$$

A seguir, mostraremos como estender estas equações para a construção `if-then-else` e para o `while`. A forma de um `if-then-else` é mostrada na Figura 2(b) e as funções são uma simples generalização das equações para o `if-then`. A função de fluxo $F_{\texttt{if-then-else}}$ construída no primeiro passo está relacionada às funções de fluxo dos seus componentes da seguinte forma:

$$F_{\texttt{if-then-else}} = (F_{\texttt{then}} \circ F_{\texttt{if/Y}}) \sqcap (F_{\texttt{else}} \circ F_{\texttt{if/N}})$$

e as funções de propagação construídas no segundo passo são:

$$\begin{aligned} in(\texttt{if}) &= in(\texttt{if-then-else}) \\ in(\texttt{then}) &= F_{\texttt{if/Y}}(in(\texttt{if})) \\ in(\texttt{else}) &= F_{\texttt{if/N}}(in(\texttt{if})) \end{aligned}$$

Para o loop `while`, temos a forma mostrada na Figura 3. No passo *bottom-up*, a função de fluxo que expressa o resultado de uma iteração do loop, voltando em seguida para a sua entrada, é $F_{\texttt{body}} \circ F_{\texttt{while/Y}}$. Assim, o resultado de fazer isto um número arbitrário de vezes é dado por:

$$F_{\texttt{loop}} = (F_{\texttt{body}} \circ F_{\texttt{while/Y}})^*$$

e o resultado da execução de todo o loop `while` é dado pela execução dos blocos `while` e `body` repetidamente, seguido pela execução do bloco `while`, saindo pelo branch `N`, isto é,

$$F_{\texttt{while-loop}} = F_{\texttt{while/N}} \circ F_{\texttt{loop}}$$

Note que no caso mais comum, ou seja, um problema de vetores de *bits*, temos que[2]

$$F_{\texttt{loop}} = (F_{\texttt{body}} \circ F_{\texttt{while/Y}})^* = id \sqcap (F_{\texttt{body}} \circ F_{\texttt{while/Y}})$$

mas que as equações acima são válidas, independentemente do problema de fluxo direto que estamos resolvendo. No passo *top-down*, temos para o loop `while`

$$\begin{aligned} in(\texttt{while}) &= F_{\texttt{loop}}(in(\texttt{while-loop})) \\ in(\texttt{body}) &= F_{\texttt{while/Y}}(in(\texttt{while})) \end{aligned}$$

Novamente, se não distinguirmos as saídas `Y` e `N`, teremos:

$$F_{\texttt{loop}} = (F_{\texttt{body}} \circ F_{\texttt{while}})^*$$

[2] Para um reticulado de funções de fluxo, $\mathbf{L}^F$, podemos dizer que, se $f \in \mathbf{L}^F$, então $f^* = id \sqcap f$.

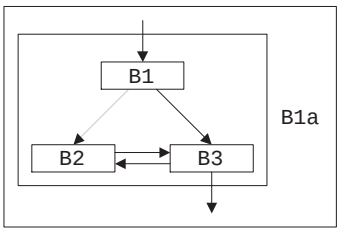


Figura 4: Uma Região Imprópria

e o resultado da execução de todo o loop `while` é dado pela execução dos blocos `while` e `body` repetidamente, seguida pela execução do bloco `while`, e saindo pelo branch `N`, isto é,

$$\begin{aligned} F_{\texttt{while-loop}} &= F_{\texttt{while}} \circ F_{\texttt{loop}} \\ &= F_{\texttt{while}} \circ (F_{\texttt{body}} \circ F_{\texttt{while}})^* \\ in(\texttt{while}) &= F_{\texttt{loop}}(in(\texttt{while-loop})) \\ in(\texttt{body}) &= F_{\texttt{while}}(in(\texttt{while})) \end{aligned}$$

A partir dos casos `if-then` e `if-then-else`, pode-se generalizar as equações para uma região acíclica genérica. Para construirmos as equações para uma região cíclica *própria*[3] qualquer, removemos os arcos de volta, obtendo uma região acíclica. A partir das equações para a região acíclica, escrevemos as equações para a região cíclica percorrendo os caminhos de volta de cada bloco dentro da região para o bloco inicial.

Para uma região acíclica imprópria, construímos, no passo *bottom-up*, um conjunto de equações semelhantes às equações de uma região acíclica genérica, que representam o efeito do fluxo de dados de um caminho a partir da entrada da região, para qualquer ponto interno. No passo *top-down*, usamos as funções construídas no passo *bottom-up* para propagar as informações de fluxo de dados para cada ponto dentro da região da maneira usual, ou seja, iniciando com as informações que temos na sua entrada. A maior diferença entre estas equações e as demais equações é que o sistema *top-down* para a região imprópria é recursivo, visto que a região contém múltiplos ciclos. Dado um sistema de equações recursivas, podemos proceder de uma das três formas abaixo:

1. Usar *splitting* de nodos, que pode transformar regiões impróprias em próprias, com possivelmente muito mais nodos.

2. Resolver o sistema recursivo de equações iterativamente, usando como dados iniciais quaisquer informações que as equações para a construção envolvente produzirem na entrada da região, cada vez que resolvemos o problema de fluxo de dados.

3. Considerar o sistema de equações um problema de fluxo de dados direto definido, não sobre L, mas sobre L^F, o reticulado de funções monotônicas de $L \to L$, e resolvê-lo, produzindo função de fluxo que correspondem aos caminhos dentro da região; isto requer que o reticulado L seja finito, o que é suficiente para a maioria dos problemas que consideramos, incluindo todos os problemas de vetores de *bits*.

Por exemplo, considere a região imprópria da Figura 4, e suponha que ela reduza para a região denominada `B1a`, como mostrado. A equação base para `B1a` é:

$$F_{\texttt{B1a}} = ((F_{\texttt{B3}} \circ F_{\texttt{B2}})^+ \circ F_{\texttt{B1}}) \sqcap ((F_{\texttt{B3}} \circ F_{\texttt{B2}})^* \circ F_{\texttt{B3}} \circ F_{\texttt{B1}})$$

[3] Uma região cíclica própria, R, é um conjunto de blocos, que formam um ciclo, tal que a única forma de acessar algum bloco pertencente a R é a partir do nodo inicial.

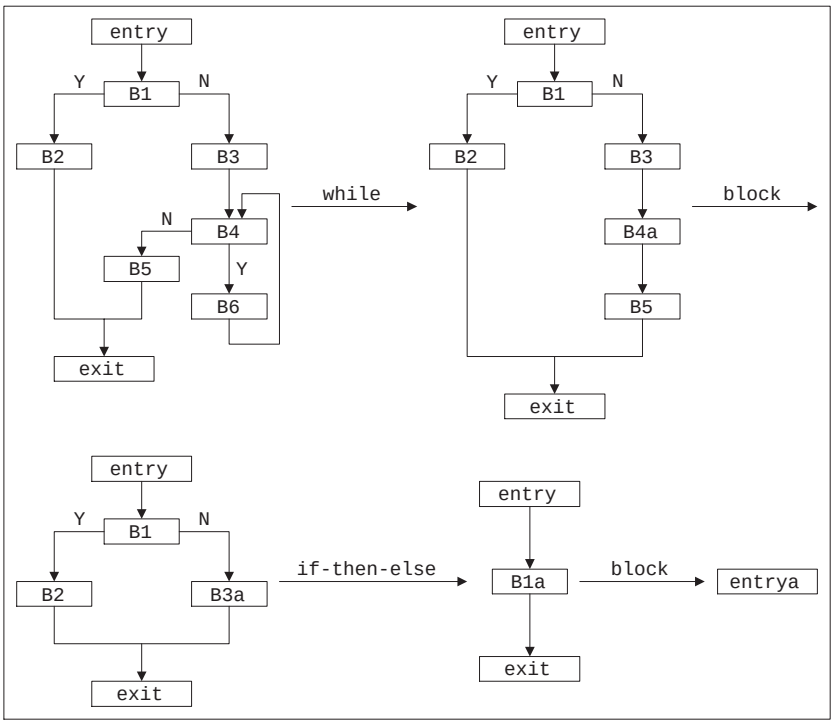


Figura 5: Análise de Fluxo de Controle Estrutural para o Exemplo de *Reaching Definitions*

Para as equações *top-down*, temos o seguinte sistema recursivo:

$$\begin{aligned} in(\texttt{B1}) &= in(\texttt{B1a}) \\ in(\texttt{B2}) &= F_{\texttt{B1}}(in(\texttt{B1})) \sqcap F_{\texttt{B3}}(in(\texttt{B3})) \\ in(\texttt{B3}) &= F_{\texttt{B1}}(in(\texttt{B1})) \sqcap F_{\texttt{B2}}(in(\texttt{B2})) \end{aligned}$$

Podemos resolver as equações para $in(\texttt{B2})$ e $in(\texttt{B3})$ no reticulado de função para produzir:

$$\begin{aligned} in(\texttt{B2}) &= (((F_{\texttt{B3}} \circ F_{\texttt{B2}})^* \circ F_{\texttt{B1}}) \sqcap ((F_{\texttt{B3}} \circ F_{\texttt{B2}})^* \circ F_{\texttt{B3}} \circ F_{\texttt{B1}}))(in(\texttt{B1})) \\ &= ((F_{\texttt{B3}} \circ F_{\texttt{B2}})^* \circ (id \sqcap F_{\texttt{B3}}) \circ F_{\texttt{B1}})(in(\texttt{B1})) \end{aligned}$$

e

$$\begin{aligned} in(\texttt{B3}) &= (((F_{\texttt{B2}} \circ F_{\texttt{B3}})^* \circ F_{\texttt{B1}}) \sqcap ((F_{\texttt{B2}} \circ F_{\texttt{B3}})^* \circ F_{\texttt{B2}} \circ F_{\texttt{B1}}))(in(\texttt{B1})) \\ &= ((F_{\texttt{B3}} \circ F_{\texttt{B2}})^* \circ (id \sqcap F_{\texttt{B2}}) \circ F_{\texttt{B1}})(in(\texttt{B1})) \end{aligned}$$

Como um exemplo de análise de fluxo de dados estrutural direta, considere o problema *reaching definitions*. Primeiramente, devemos fazer a análise de fluxo de controle estrutural para o problema, como mostrado na Figura 5. A primeira equação que construímos no passo *bottom-up* da análise de fluxo de dados é, para o loop `while`, a seguinte[4]:

$$F_{\texttt{B4a}} = F_{\texttt{B4}} \circ (F_{\texttt{B6}} \circ F_{\texttt{B4}})^* = F_{\texttt{B4}} \circ (id \sqcap (F_{\texttt{B6}} \circ F_{\texttt{B4}}))$$

Para o `if-then-else` reduzido para `B1a`:

$$F_{\texttt{B1a}} = (F_{\texttt{B2}} \circ F_{\texttt{B1}}) \sqcap (F_{\texttt{B3}} \circ F_{\texttt{B1}})$$

[4] Note que não precisamos distinguir as saídas `Y` e `N`, por isso elas foram omitidas.

$F_{\texttt{entry}}$	$=$	id
$F_{\texttt{B1}}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle)$	$=$	$\langle 111x_4x_500x_8\rangle$
$F_{\texttt{B2}}$	$=$	id
$F_{\texttt{B3}}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle)$	$=$	$\langle x_1x_2x_31x_5x_6x_70\rangle$
$F_{\texttt{B4}}$	$=$	id
$F_{\texttt{B5}}$	$=$	id
$F_{\texttt{B6}}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle)$	$=$	$\langle x_10001111\rangle$

Tabela 1: Funções de Fluxo para o Grafo de Fluxo da Figura 5

e para o bloco reduzido para `entrya`:

$$F_{\texttt{entrya}} = F_{\texttt{exit}} \circ F_{\texttt{B1a}} \circ F_{\texttt{entry}}$$

No passo *top-down*, construímos, para os componentes do bloco `entrya`, as equações:

$$\begin{aligned} in(\texttt{entry}) &= Init \\ in(\texttt{B1a}) &= F_{\texttt{entry}}(in(\texttt{entry})) \\ in(\texttt{exit}) &= F_{\texttt{B1a}}(in(\texttt{B1a})) \end{aligned}$$

Para o `if-then-else` reduzido para `B1a`:

$$\begin{aligned} in(\texttt{B1}) &= in(\texttt{B1a}) \\ in(\texttt{B2}) &= in(\texttt{B3}) = F_{\texttt{B1}}(in(\texttt{B1a})) \end{aligned}$$

Para o bloco reduzido para `B3a`:

$$\begin{aligned} in(\texttt{B3}) &= in(\texttt{B3a}) \\ in(\texttt{B4a}) &= F_{\texttt{B3}}(in(\texttt{B3a})) \\ in(\texttt{B5}) &= F_{\texttt{B4a}}(in(\texttt{B4a})) \end{aligned}$$

Para o loop `while` reduzido para `B4a`:

$$\begin{aligned} in(\texttt{B4}) &= (F_{\texttt{B6}} \circ F_{\texttt{B4}})^*(in(\texttt{B4a})) = (id \sqcap (F_{\texttt{B6}} \circ F_{\texttt{B4}}))(in(\texttt{B4a})) \\ in(\texttt{B6}) &= F_{\texttt{B4}}(in(\texttt{B4})) \end{aligned}$$

O valor inicial de $in(\texttt{entry})$ e a função de fluxo para os blocos individuais são dados na Tabela 1. Nosso primeiro passo na resolução das equações para os valores de $in()$, à mão, é para simplificar a equação para as funções de fluxo compostas:

$$\begin{aligned} F_{\texttt{B4a}} &= id \sqcap F_{\texttt{B6}} \\ F_{\texttt{B3a}} &= F_{\texttt{B4a}} \circ F_{\texttt{B3}} \\ F_{\texttt{B1a}} &= F_{\texttt{B1a}} \sqcap (F_{\texttt{B3a}} \circ F_{\texttt{B1}}) \end{aligned}$$

A partir daí, computamos o valor de $in(\,)$, começando a partir de $in(\texttt{entry})$ e usando os valores disponíveis das funções $F_B(\,)$, o que resulta nos valores mostrados na Tabela 2.

4 Análise de Intervalos

Agora que já construímos todos os mecanismos para fazer a análise de fluxo de dados estrutural, fazer a análise de intervalos é trivial, pois é idêntica à análise estrutural, exceto que apenas três tipos de regiões aparecem: acíclica geral, própria e imprópria.

$in(\texttt{entry})$	=	$\langle 00000000 \rangle$
$in(\texttt{B1})$	=	$\langle 00000000 \rangle$
$in(\texttt{B2})$	=	$\langle 11100000 \rangle$
$in(\texttt{B3})$	=	$\langle 11100000 \rangle$
$in(\texttt{B4})$	=	$\langle 11111111 \rangle$
$in(\texttt{B5})$	=	$\langle 11111111 \rangle$
$in(\texttt{B6})$	=	$\langle 11111111 \rangle$
$in(\texttt{exit})$	=	$\langle 11111111 \rangle$

Tabela 2: Valores de $in(\ )$ Computados pela Análise Estrutural para o Exemplo de *Reaching Definitions*

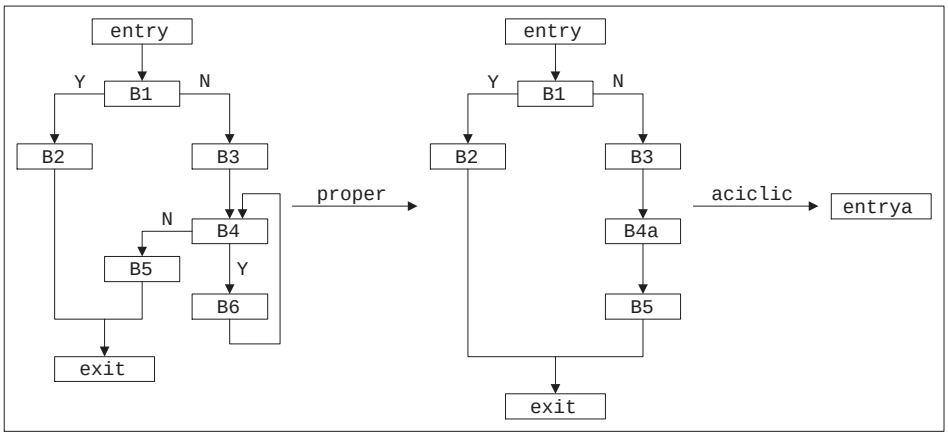


Figura 6: Análise de Fluxo de Controle por Intervalos para o Exemplo de *Reaching Definitions*

Como exemplo, considere o grafo de fluxo da Figura 6, com sua redução para intervalos. O primeiro passo transforma o loop B4-B6, comprimindo-o no nodo B4a e o segundo passo reduz toda a estrutura acíclica resultante no nodo entrya. As funções de fluxo de dados direto correspondentes são:

$$\begin{aligned} F_{\texttt{B4a}} &= F_{\texttt{B4}} \circ (F_{\texttt{B6}} \circ F_{\texttt{B4}})^* = id \sqcap F_{\texttt{B6}} \\ F_{\texttt{entrya}} &= F_{\texttt{exit}} \circ (F_{\texttt{B2}} \sqcap (F_{\texttt{B5}} \circ F_{\texttt{B4a}} \circ F_{\texttt{B3}})) \circ F_{\texttt{B1}} \circ F_{\texttt{entry}} \end{aligned}$$

e as equações para $in()$ são:

$$\begin{aligned} in(\texttt{entry}) &= in(\texttt{entrya}) = Init \\ in(\texttt{B1}) &= F_{\texttt{entry}}(in(\texttt{entry})) \\ in(\texttt{B2}) &= F_{\texttt{B1}}(in(\texttt{B1})) \\ in(\texttt{B3}) &= F_{\texttt{B1}}(in(\texttt{B1})) \\ in(\texttt{B4a}) &= F_{\texttt{B3}}(in(\texttt{B3})) \\ in(\texttt{B4}) &= in(\texttt{B4a}) \sqcap F_{\texttt{B6}}(in(\texttt{B4a})) \\ in(\texttt{B6}) &= F_{\texttt{B4}}(in(\texttt{B4})) = in(\texttt{B4}) \\ in(\texttt{B5}) &= F_{\texttt{B4a}}(in(\texttt{B4a})) = in(\texttt{B4a}) \sqcap F_{\texttt{B6}}(in(\texttt{B4a})) \\ in(\texttt{exit}) &= F_{\texttt{B2}}(in(\texttt{B2})) \sqcap F_{\texttt{B5}}(in(\texttt{B5})) \end{aligned}$$

Os valores computados serão idênticos aos computados pela análise estrutural.

```
x ← a[i]      x ← access(a,i)
a[j] ← 4      a ← update(a,j,4)
(a)           (b)
```

Figura 7: Atribuições Envolvendo Elementos de Arranjos e sua Tradução para a Forma *access/update*

5 Outras Abordagens

Uma outra abordagem para análise de fluxo de dados é a chamada *slotwise analysis*. Nesta abordagem, ao invés de termos grandes vetores de *bits* para representarmos uma característica de fluxo de dados de variáveis ou algum outro tipo de construção e operar sobre estes vetores de *bits*, consideramos cada *slot* de todos os vetores de *bits* separadamente. Isto é, primeiro consideramos o que acontece com o primeiro *slot* em todos os vetores de *bits* no procedimento, em seguida com o segundo, e assim por diante. Para alguns problemas de fluxo de dados, esta abordagem é inútil, visto que eles dependem de combinar informações de diferentes *slots* em dois ou mais vetores de *bits* para computar o valor de um *slot* em outro vetor de *bits*. Entretanto, para muitos problemas, tal como *reaching definitions* e expressões disponíveis, cada *slot* em um endereço particular em um procedimento depende somente daquele *slot* em outro endereço. Para o problema das expressões disponíveis, por exemplo, as informações na maioria dos *slots* é o valor default 0 (= não disponível), na maioria dos endereços. Tal combinação pode tornar esta abordagem muito atrativa.

A aplicação desta abordagem ao problema da análise de redundância parcial é mostrada em [2]. Esta análise é muito importante e é utilizada em muitos compiladores comerciais.

6 Lidando com Arranjos, Estruturas e Ponteiros

Até agora, não lidamos, na análise de fluxo de dados, com valores mais complexos que constantes e variáveis simples, cujos valores são restritos a tais constantes. Visto que variáveis e, em algumas linguagens, constantes podem ter também arranjos, registros e ponteiros como valores, é essencial que consideremos como ajustar tais elementos no esquema de análise de fluxo de dados.

Uma opção que é usada em muitos compiladores é simplesmente ignorar atribuições a arranjos e registros e tratar atribuições via ponteiros de maneira pessimista. Nesta abordagem, assume-se que um ponteiro pode apontar para qualquer valor de variável e, portanto, uma atribuição via ponteiros pode implicitamente afetar alguma variável. Linguagens como Pascal provêem certa facilidade neste ponto, pois restringe os ponteiros a apontarem somente para objetos dos tipos declarados como ponteiros. Lidar com ponteiros de maneira que possam ser produzidas informações úteis requer o que chamamos de *análise de alias*. Ponteiros na *heap* podem ser modelados de maneira conservativa, considerando a *heap* como um único objeto, como, por exemplo, um arranjo, onde assumimos que qualquer atribuição via ponteiro acessa ou muda algum objeto na *heap*.

Em C, ponteiros não são restritos a apontar para objetos na *heap*, pois podem apontar para objetos na pilha ou para objetos alocados estaticamente. Por isso, os métodos de análise de alias são muito importantes para otimizações agressivas de programas escritos em C.

Algumas linguagens permitem atribuições a arranjos nos quais os valores de todos os seus elementos são atualizados de uma vez só. Tais atribuições podem ser manipuladas facilmente, considerando variáveis e constantes arranjos como variáveis e constantes comuns. Entretanto, a maior parte das atribuições a arranjos atualizam somente um elemento, como em `A[3] ← 5` ou `A[i] ← 2`. Atribuições que atualizam um elemento conhecido podem ser tratadas como atribuições comuns, mas isto ainda não pode ser aplicado à maioria das operações com arranjos. Uma possibilidade para lidar com atribuições que atualizam um elemento do arranjo endereçado por uma variável é traduzi-las para uma forma que utiliza atribuições de acesso (*access*) e de atualização (*update*), que as fazem parecer operar sobre todo o arranjo, como na Figura 7. Apesar de tais operadores permitirem que algoritmos de fluxo de dados funcionem corretamente, eles

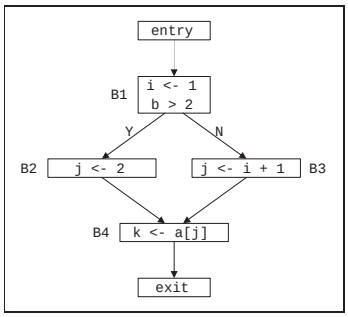


Figura 8: Exemplo Simples de Análise de Propagação de Constantes

geralmente produzem informações muito ingênuas para serem úteis na otimização de operações com arranjos. Uma alternativa comum é fazer a análise de dependência de operações com arranjos, durante a fase de *análise de dependência*. Tal abordagem pode prover informações mais precisas sobre arranjos, mas ao custo de muito mais computações.

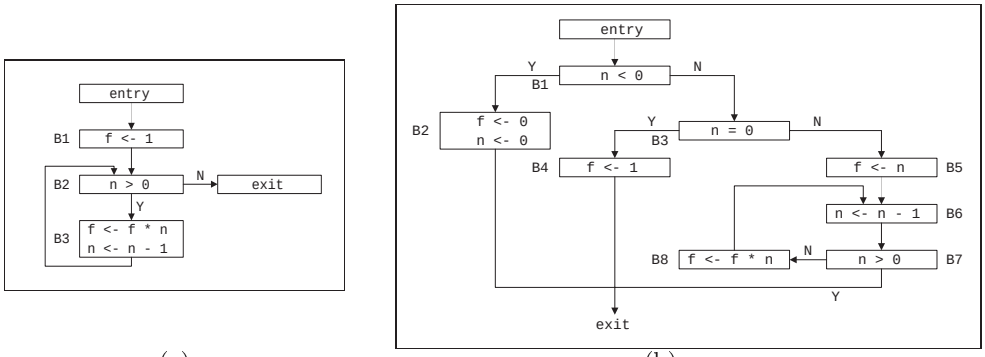
Na maioria das linguagens, atribuições envolvendo referências diretas a elementos de registros sem o uso de ponteiros podem usar somente nomes de membros, que são constantes. Desta forma, atribuições a registros podem ser tratadas como acessos e atualizações ao registro todo, como sugerido para arranjos, ou então podem ser tratadas como lidar com membros individuais. A última abordagem pode resultar em otimizações mais efetivas, se os registros forem utilizados freqüentemente no programa. Se a linguagem fonte permitir que variáveis sejam usadas para selecionar membros de registros, então eles são tratados como arranjos de tamanho fixo com nomes de elementos simbólicos e podem ser manipulados como arranjos.

7 Análises Mais Ambiciosas

Até agora, consideramos somente formas relativamente ingênuas de análise de fluxo de dados. Nesta seção, exploraremos como a complexidade dos reticulados utilizados e o poder de raciocínio sobre as operações realizadas afetam as propriedades que podemos determinar.

Considere a análise de propagação de constantes sobre o exemplo simples da Figura 8. Se as operações aritméticas, tais como, `i + 1`, são todas consideradas não interpretadas, isto é, se assumirmos que não temos informações sobre o seu efeito, então não temos meios de determinar que o valor de `j` é constante na entrada de `B4`. Se, por outro lado, fortalecermos nossa análise de propagação de constantes para incluir a habilidade de fazer adição de constantes, então podemos facilmente determinar que `j` tem o valor 2 na entrada de `B4`.

No exemplo da Figura 9(a), assumindo que podemos raciocinar sobre a subtração de 1 e a comparação com zero e que distinguimos as saídas `Y` e `N` dos testes, então podemos concluir que na entrada do bloco `exit`, o valor de `n` é menor ou igual a zero. Se estendermos este programa para o mostrado na Figura 9(b), então podemos concluir que `n = 0` na entrada do bloco `exit`. Mais ainda, se pudermos raciocinar sobre funções inteiras, então podemos determinar que, no mesmo ponto, se $n_0 \geq 0, f = n_0!$, onde n_0 representa o valor de `n` na entrada do fluxo de dados. Isto, pelo menos, sugere que podemos utilizar técnicas analíticas de fluxo de dados na verificação de programas. Para isso, as informações de fluxo de dados que precisamos associar às saídas de cada bloco são as asserções indutivas mostradas na Tabela 3. Mesmo que isto requeira mais cálculos e tenha uma complexidade computacional muito maior que qualquer análise que podemos realmente utilizar em um compilador, isto, ao menos, demonstra o espectro de possibilidades da Análise de Fluxo de Dados.



(a) (b)

Figura 9: Computação do Fatorial: (a) Simplificada e (b) Completa.

Bloco	**Asserção Indutiva**
`entry`	$\mathtt{n} = n_0$
`B1/Y`	$\mathtt{n} = n_0 < 0$
`B1/N`	$\mathtt{n} = n_0 \geq 0$
`B2`	$\mathtt{n} = n_0$ e $\mathtt{f} = n_0!$
`B3/Y`	$\mathtt{n} = n_0 = 0$ e $\mathtt{f} = 1$
`B3/N`	$\mathtt{n} = n_0 > 0$
`B4`	$\mathtt{n} = n_0 = 0$ e $\mathtt{f} = n_0!$
`B5`	$\mathtt{n} = n_0 > 0$ e $\mathtt{f} = n_0$
`B6`	$\mathtt{n} \geq 0$ e $\mathtt{f} = n_0 \times (n_0 - 1) \times \cdots \times (\mathtt{n} + 1)$
`B7/Y`	$\mathtt{n} > 0$ e $\mathtt{f} = n_0 \times (n_0 - 1) \times \cdots \times (\mathtt{n} + 1)$
`B7/N`	$\mathtt{n} = 0$ e $\mathtt{f} = n_0 \times (n_0 - 1) \times \cdots \times 1 = n_0!$
`B8`	$\mathtt{n} = n_0 = 0$ e $\mathtt{f} = n_0 \times (n_0 - 1) \times \cdots \times \mathtt{n}$

Tabela 3: Asserções indutivas associadas às saídas de cada bloco da Figura 9(b), que são necessárias para determinar que ela computa a função inteira fatorial

8 Transformações Utilizando Análise de Fluxo de Dados

Nesta seção mostraremos como um compilador otimizador pode fazer transformações para melhoria do código a partir dos resultados da análise de fluxo de dados.

8.1 Eliminação de Sub-Expressões Comuns

Dado um comando no grafo de fluxo da forma $s : t \leftarrow x \oplus y$, onde a expressão $x \oplus y$ está *disponível* em s, a computação dentro de s pode ser eliminada.

Algoritmo: Determine as *expressões disponíveis*, isto é, encontre comandos da forma $n : v \leftarrow x \oplus y$, tal que o caminho a partir de n até s não computa $x \oplus y$ nem define x ou y. Escolha um novo temporário w, e para cada comando n, rescreva-o da forma:

$$n : w \leftarrow x \oplus y$$
$$n' : v \leftarrow w$$

Finalmente, modifique o comando s para

$$s : t \leftarrow w$$

Podemos utilizar propagação de cópias para remover alguma, ou até mesmo todas, as quádruplas extras de atribuição.

8.2 Propagação de Constantes

Suponha que temos um comando $d : t \leftarrow c$, onde c é uma constante, e outro comando n que usa t, tal que $n : y \leftarrow t \oplus x$. Nós sabemos que t é constante em n se d alcança n e nenhuma outra definição de t alcança n. Neste caso, podemos reescrever n como $y \leftarrow c \oplus x$.

8.3 Propagação de Cópias

Esta otimização é semelhante à propagação de constantes, mas, ao invés de uma constante c, temos uma variável z.

Suponha que temos um comando $d : t \leftarrow z$, e outro comando n que utiliza t, tal que $n : y \leftarrow t \oplus x$. Se d alcança e nenhuma outra definição de t alcança n e não há definição de z em algum caminho a partir de d até n, incluindo um caminho que passa por n uma ou mais vezes, então rescreva n como $n : y \leftarrow z \oplus x$.

Se fizermos propagação de cópia antes da alocação de registradores, podemos aumentar o número de derramamentos para a memória. Portanto, se nossa única razão para fazermos a propagação de cópia é remover instruções `MOVE` redundantes, então é melhor esperar até a alocação de registradores. Entretanto, propagação de cópia no código intermediário pode habilitar o reconhecimento de outras otimizações, tais como a eliminação de sub-expressões comuns. Por exemplo, no programa

$$\begin{aligned} a &\leftarrow y + z \\ u &\leftarrow y \\ c &\leftarrow u + z \end{aligned}$$

as duas expressões de soma não são reconhecidas como sub-expressões comuns, a menos que seja feita a propagação de cópia de $u \leftarrow y$.

8.4 Eliminação de Código Morto

Se houver uma quádrupla $s : a \leftarrow b \oplus c$ ou $s : a \leftarrow M[x]$, tal que a não está viva à saída de s, então a quádrupla pode ser eliminada.

Algumas instruções têm efeitos colaterais implícitos. Por exemplo, se a máquina estiver configurada para levantar uma exceção em uma divisão por zero, então a remoção de uma instrução que poderia causar uma exceção mudará o resultado da computação.

O compilador nunca deveria fazer uma modificação que altere o comportamento do programa, mesmo se a mudança parecer benigna, tal como remover um "erro"de execução. O problema de tais otimizações é que o programador não pode predizer o comportamento do programa – e um programa depurado com o otimizador habilitado pode falhar com o otimizador desabilitado.

9 Conclusões

Neste trabalho, abordamos os principais aspectos de Análise de Fluxo de Dados, que tem por finalidade, fornecer informações que possibilitarão ao compilador realizar transformações que produzem um código de maior qualidade para o programa fonte. Mostramos a taxonomia e os principais problemas de fluxo de dados para Otimização de Código em Compiladores, mostramos as principais técnicas de escrita das equações de fluxo de dados, que resolvidas, nos dão as informações que precisamos. Além disso, vimos que a análise de fluxo de dados não é suficiente para lidar com arranjos e ponteiros, o que torna necessárias as Análises de Dependências e de Alias. Apesar desta limitação, a Análise de Fluxo de Dados é importante para um grande número de otimizações, como as que mostramos nos exemplos da Seção 8.

Referências

1. Appel, A.W. *Modern Compiler Implementation in Java – Basic Techniques.* pp. 333–351. Cambridge University Press. Cambridge, UK. 1997.

2. Dhamdhere, Dhananjay M. Barry K. Rosen, and F. Kenneth Zadeck. How to Analyze Large Programs Efficiently and Informatively. In *PLDI'94.* pp. 36–46.

3. Muchnick, S.S. *Advanced Compiler Design and Implementation.* pp. 217–266. Morgan Kaufmann Publishers. San Francisco, USA. 1997.

3 Análise de Fluxo de Dados

Análise de Dependência

Marco Túlio de Oliveira Valente

Análise de Dependência

Marco Túlio de O. Valente

Junho de 1999

1 Introdução

O objetivo da análise de dependência é identificar relações de precedência que devem ser satisfeitas entre as instruções de um bloco básico ou de uma unidade de código maior a fim de garantir a execução correta de seu código. Portanto, é uma ferramenta imprescindível para escalonamento de instruções e para as otimizações de *caches* de dados.

Neste trabalho, usaremos a notação $S_1 \triangleleft S_2$ para indicar que a execução da instrução S_1 precede a execução de S_2 em um determinado fragmento de código. Uma *dependência* entre duas instruções de um programa é uma relação que restringe a ordem de execução das mesmas. Dependências podem ser de dois tipos: de controle ou de dados. *Dependências de controle* são derivadas do fluxo de controle do programa. Como exemplo, temos a relação entre S_2 e S_3 no fragmento de código abaixo, no qual a execução de S_2 sempre precederá a de S_3. Já as *dependências de dados* são derivadas do fluxo de dados de um programa, podendo ser subclassificadas em quatro tipos:

Dependência Verdadeira: Ocorre quando $S_1 \triangleleft S_2$ e S_1 produz um valor que será usado por S_2. Como exemplo, temos a dependência entre S_3 e S_4 abaixo, onde S_3 produz um valor d que S_4 usa logo em seguida.

Antidependência: Ocorre quando $S_1 \triangleleft S_2$ e S_1 usa um valor que será produzido mais adiante por S_2. Como exemplo, temos novamente a dependência entre S_3 e S_4 abaixo, onde S_3 usa o valor e que será produzido em seguida por S_4.

Dependência de Saída: Ocorre quando $S_1 \triangleleft S_2$ e tanto S_1 como S_2 produzem o mesmo valor. Como exemplo, temos a dependência entre S_3 e S_5 abaixo, onde ambas instruções produzem o valor d.

Dependência de Entrada: Ocorre quando $S_1 \triangleleft S_2$ e tanto S_1 como S_2 usam o mesmo valor. Como exemplo, temos novamente a dependência entre S_3 e S_5 abaixo, onde ambas instruções usam o valor e. Observe que esta modalidade de dependência de dados não impõe nenhuma restrição na ordem de execução das instruções S_1 e S_2.

S_1 a:= b + c
S_2 if a > 10 goto L1
S_3 d:= b * e

S_4 e:= d + 1
S_5 L_1: d:= e / 2

As relações de dependência existentes entre as instruções de um fragmento de código podem ser representadas por meio de um grafo dirigido, chamado *grafo de dependência*, onde os nós representam instruções e as arestas representam dependências. Cada aresta é etiquetada com o tipo de dependência que representa, exceto no caso de dependências de fluxo, cujas arestas não são normalmente etiquetadas. A Figura 1 mostra o grafo de dependência para o bloco de código acima. Dependências de controles também são geralmente omitidas em grafos de dependência, a não ser que tal dependência seja a única que conecte dois nós. No exemplo, há uma dependência de controle entre S_2 e S_4, mas ela é omitida no grafo.

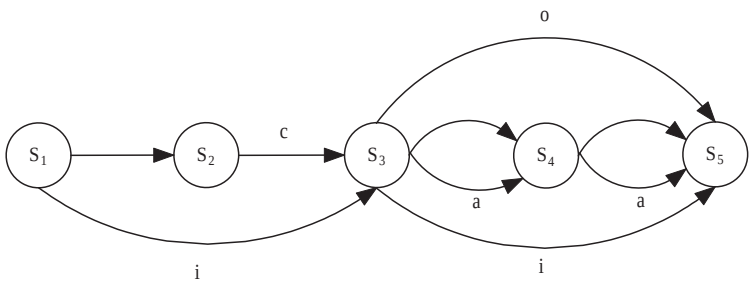


Figura 1: Grafo de Dependência para o Bloco de Código Mostrado

2 Grafos de Dependência de Blocos Básicos

No caso de o fragmento de código analisado ser um bloco básico, o grafo de dependência será um grafo dirigido acíclico (DAG), chamado de *DAG de dependência*. Os nós de um DAG de dependência representam instruções de máquina ou de nível intermediário. Uma aresta do nó I_1 para o nó I_2 representa uma dependência de dados, classificada como verdadeira, antidependência ou de saída, ou então uma das seguintes situações:

- Quando não se pode determinar se I_1 pode ser movido para antes de I_2, como, por exemplo, no caso de I_1 ser uma instrução que lê o endereço de memória `[r1](4)` e I_2 uma instrução que grava um valor em `[r2+12](4)`. Neste caso, a não ser que saibamos que `r2+12` e `r1` se referem a um mesmo endereço, deveremos assumir que há uma dependência entre I_1 e I_2.
- Quando I_1 e I_2 dão origem a um conflito estrutural.

Em um DAG de dependência, uma aresta de I_1 para I_2 é etiquetada com a *latência* existente entre as duas instruções, isto é, o número de ciclos necessários entre o disparo de I_1 e de I_2, menos o número de ciclos em que I_1 executa sem que nenhuma outra instrução seja disparada.

Este valor normalmente é igual a um, mas pode também ser zero, como no caso de máquinas superescalares. Por exemplo, se I_2 pode ser disparada no ciclo seguinte ao disparo de I_1, então a latência é zero.

O algoritmo que utilizaremos para construir um DAG de dependência de um bloco básico fará uso das seguintes funções:

- Conflito (I_1, I_2): função booleana que retorna verdadeiro se a execução de I_1 deve preceder a de I_2 e falso, caso contrário.
- Latência (I_1, n_1, I_2, n_2): função que retorna o número de ciclos de latência existentes entre o n_2-ésimo ciclo da instrução I_2, supondo-se que esteja sendo executado neste momento o n_1-ésimo ciclo da instrução I_1.

O método mais simples para calcular a função latência é baseado no uso de *vetores de recursos*, isto é, vetores que representam os recursos utilizados pela instrução em cada um de seus ciclos. Por exemplo, as instruções `add.s` e `mul.s` em uma arquitetura MIPS possuem os vetores de recursos mostrados abaixo, onde os recursos são abreviados pelas seguintes letras: A (*mantissa add*), M (*multiplier first stage*), N (*multiplier second stage*), R (*adder round*), S (*operand shift*) e U (*unpack*).

	1	2	3	4	5	6	7
`add.d`	U	S,A	A,R	R,S			
`mul.s`	U	M	M	M	N	N,A	R

A implementação da função latência para duas instruções é feita obtendo-se primeiro seus respectivos vetores de recursos e deslocando então o vetor de recursos da instrução I_2 para direita sempre que for detectado um conflito. Por exemplo, uma chamada da forma *latência (mul.s, 4, add.s, 1)* daria origem a dois deslocamentos para direita no vetor de recursos de `add.s`, como mostrado abaixo. Logo, o resultado desta chamada seria dois.

	4	5	6	7	8	9
`mul.s`	M	N	N,A	R		
`add.s`	U	S,A	stall	stall	A,R	R

De posse da implementação das duas funções já mencionadas, um possível algoritmo para construir o DAG de depedência seria o seguinte:

Entrada: sequência de instruções `inst [1..m]` de um bloco básico
Saída: DAG de dependência $G = (V, A)$
para j:= 1 **até** m **faça**
    V:= V ∪ {j}
    para k:= 1 **até** j-1 **faça**
        se conflito (inst [k], inst [j]) **então**
            A:= A ∪ { (k, j, latência (inst [k], 1, inst [j], 2) }

Este algoritmo possui complexidade $O(n^2)$, já que toda nova instrução é comparada com cada uma das instruções anteriores.

3 Dependência em *Loops*

Dependência em *loops* procura analisar as dependências existentes entre variáveis indexadas utilizadas no corpo de *loops*, sendo novamente este estudo importante para as otimizações de *caches* de dados. Para determinação de dependências de *loops* são considerados apenas laços na forma canônica mostrada abaixo:

for i_1:= 1 **to** n_1 **do**
    for i_2:= 1 **to** n_2 **do**
        .....
            for i_k:= 1 **to** n_k **do**
                comandos

As relações de dependência entre variáveis indexadas de *loops* são mais complexas do que entre variáveis escalares, pois são funções não apenas da ordem dos comandos, mas também dos índices das variáveis. A notação $S_1[i_{11}, \ldots, i_{k1}] \rightarrow S_2[i_{12}, \ldots, i_{k2}]$ denota que $S_1[i_{11}, \ldots, i_{k1}]$ é executado antes que $S_2[i_{12}, \ldots, i_{k2}]$, onde $i_1, \ldots, i_k$ correspondem aos índices dos *loops*, do mais externo para o mais interno.

Seja o seguinte exemplo:

```
        for i1:= 1 to 3 do
            for i2:= 1 to 4 do
S1              t:= x + y;
S2              a [i1, i2]:= b [i1, i2] + c [i1, i2];
S3              b [i1, i2]:= a [i1, i2-1] * d[i1+1, i2] + t;
            endfor
        endfor
```

Temos neste exemplo que $S_2[i_1, i_2 - 1] \rightarrow S_3[i_1, i_2]$ e que $S_2[i_1, i_2] \rightarrow S_3[i_1, i_2]$.

Uma dependência de *loop* pode ser de dois tipos:

Loop-independent: quando é independente dos laços que a cercam. Por exemplo a dependência entre $b[i_1, i_2]$ de S_2 e o mesmo valor em S_3 no exemplo acima, pois mesmo que os dois comandos não fizessem parte de um laço, essa antidependência existiria.

Loop-carried: quando existe devido aos laços em que está incluída. Por exemplo, a dependência devida ao fato de S_2 armazenar um valor em a que será usado por S_3 na próxima iteração do laço mais interno.

Estes dois conceitos de dependência são usados em otimizações que tentam substituir o acesso a vetores por acesso a escalares, que podem por exemplo estar alocados em registradores.

4 Testes de Dependência

Uma das principais estratégias utilizadas pelas otimizações de *caches* consiste em transformar o aninhamento de *loops* de forma a obter padrões de acesso a memória que preservem a localidade no acesso aos dados. Para isso, é essencial determinar as dependências existentes em um laço.

Seja por exemplo o seguinte laço:

```
for i:= 1 to 4 do
    b [i]:= a[3*i-5] +2;
    a [2*i+1]:= 1 / i;
endfor
```

Dependências nos acessos ao vetor a podem ocorrer na mesma iteração ou em iterações diferentes, como descrito a seguir:

- Dependências na mesma iteração: nesse caso, devemos determinar se existe um valor i tal que $2 * i + 1 = 3 * i - 5$, para $1 \leq i \leq 4$. Como não existe tal i, temos que não há dependências em uma mesma iteração (chamadas de dependências com distância zero).
- Dependências entre iterações diferentes do laço: nesse caso, devemos determinar se existem valores i_1 e i_2 tais que $2 * i_1 + 1 = 3 * i_2 - 5$, para $1 \leq i_1, i_2 \leq 4$. No caso, $i_1 = 3$ e $i_2 = 4$ satisfaz todas condições e, portanto, o laço apresenta uma dependência entre iterações diferentes.

Em geral, a determinação de dependências entre iterações de laços aninhados é um problema equivalente ao problema de programação inteira, o qual é sabido ser NP-Completo. No entanto, alguns algoritmos podem ser usados para resolver o problema quando são admitidas algumas restrições no formato geral das equações. A restrição mais comum é exigir que os subscritos sejam expressões lineares de um único índice. Assumindo esta restrição, os laços terão a seguinte forma:

for i_1:= 1 **to** h_{i1} **do**
  for i_2:= 1 **to** h_{i2} **do**
    .....
      for i_n:= 1 **to** h_{in} **do**
        .....
        $\cdots x[\ldots, a_0 + a_1 * i_1 + \cdots + a_n * i_n, \ldots] \cdots$
        $\cdots x[\ldots, b_0 + b_1 * i_1 + \cdots + b_n * i_n, \ldots] \cdots$
        .....
      endfor
    .....
  endfor
endfor

Um dos primeiros testes propostos para determinação de dependências foi o Teste do Máximo Divisor Comum (MDC), o qual é usado até hoje. O Teste do MDC utiliza a seguinte expressão:

$$\gcd\left(\bigcup_{j=1}^{n} \text{sep}(a_j, b_j, j)\right) \asymp \sum_{j=0}^{n}(a_j - b_j)$$

onde gcd é a função máximo divisor comum, $a \asymp b$ significa que a não divide b e

$$\text{sep}(a_j, b_j, j) = \begin{cases} \{a - b\} & \text{se a direção de } j \text{ é } = \\ \{a, b\} & \text{caso contrário} \end{cases}$$

Fornecidos dois acessos a um vetor, se para quaisquer índices a relação acima se verificar, então os acessos são independentes. Veja que se a relação não se verificar, não podemos afirmar se os acessos são ou não independentes.

Para o exemplo mostrado anteriormente, o teste em uma mesma iteração equivale a verificar se

$$\gcd(3 - 2) \asymp (-5 - 1 + 3 - 2)$$

que é equivalente a testar se $1 \asymp -5$, o que que é falso. Logo, não podemos afirmar que os acessos são independentes.

Em [2] são citados diversos outros métodos que já foram propostos para teste de dependência, como, por exemplo, teste MDC estendido, teste SIV (*single index variable*) forte e fraco, teste delta, teste acíclico, teste de potência etc.

5 Grafo de Dependência de Programas

Grafos de Dependência de Programas (PDG) são uma forma de código intermediário projetada para uso em otimização. O PDG de um programa consiste de um grafo de dependência de controle (CDG) e um grafo de dependência de dados. Os nós de um PDG podem ser blocos básicos, comandos, operadores individuais ou construções de um nível intermediário dentre estas citadas. O grafo de dependência de dados já foi na Seção 2.

Já o grafo de dependência de controle (CDG) é um DAG que possui predicados como raiz e nós intermediários e possui não-predicados nas folhas. Uma folha é executada se os predicados do caminho que leva da raiz até ela são satisfeitos.

Seja $G = (N, E)$ o grafo de fluxo de um procedimento. Um nó m pós-domina um nó n, m *pdom* n, se e somente se todo caminho de n até *exit* passa por m.

Um nó n é *dependente por controle* de um nó m se e somente se:

- Existe um caminho de fluxo de controle de m até n tal que todo nó neste caminho é pós-dominado por n.
- n não pós-domina m.

Para construir o CDG pode-se usar o seguinte algoritmo:

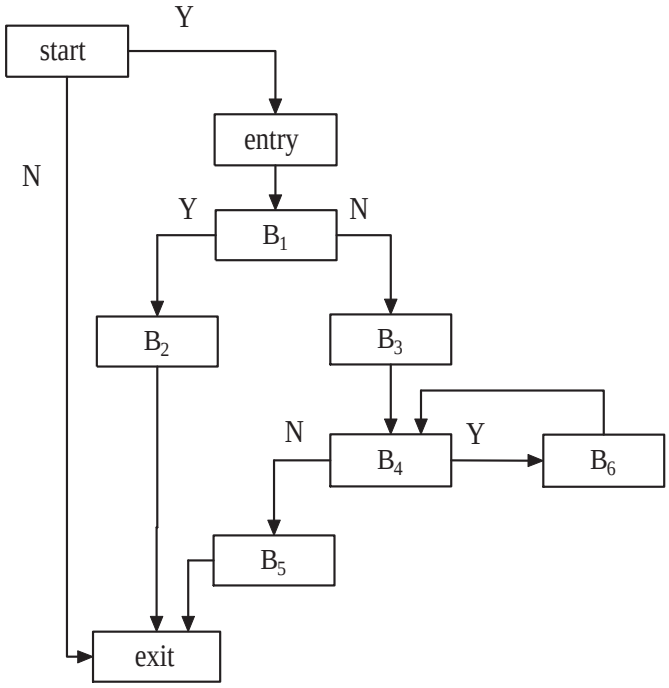


Figura 2: Grafo de Fluxo com a Adição do nó *start*

1. Adicionar um nó *start* ao grafo de fluxo com uma aresta "Y" para o nó *entry* e uma aresta "N" para o nó *exit*. O grafo resultante será chamado de G^+. Um grafo de fluxo exemplo já com o nó *start* adicionado é mostrado na figura 2.

2. Construir a relação de pós-dominância de G^+. Esta relação para o nosso grafo exemplo é mostrada na Figura 3.

3. Seja S o conjunto de arestas $m \rightarrow n$ de G^+ tais que n não pós-domina m. Para o nosso exemplo, $S = \{start \rightarrow entry, B_1 \rightarrow B_2, B_1 \rightarrow B_3, B_4 \rightarrow B_6\}$.

4. Para cada aresta $m \rightarrow n \in S$, determine o menor ancestral comum l de m e n na árvore de pós-dominância. Então todos os nós de l até n na árvore de pós-dominância serão dependentes por controle de m. O grafo resultante é chamado de CDG básico e é mostrado na Figura 4.

É comum também agrupar em "regiões" todos os nós que possuem dependências de controle derivadas de um único predicado, dando origem assim a predicados com no máximo dois sucessores. O resultado da adição de nós de agrupamento em nosso exemplo é mostrado na figura 5.

Uma importante finalidade dos PDGs é indicar quais nós podem ter sua execução paralelizada desde que não haja dependência de dados entre eles. No nosso exemplo, B_3, B_4 e B_5 podem ser executados em paralelo.

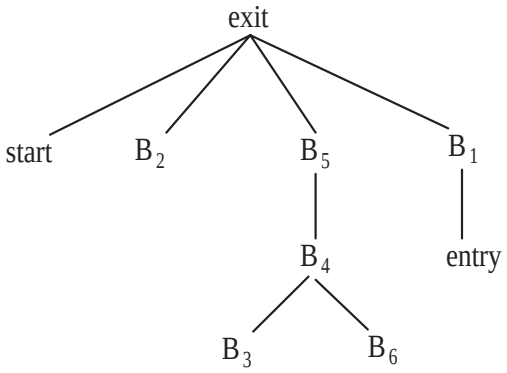


Figura 3: Árvore de Pós-dominância do Grafo de Fluxo da Figura 2

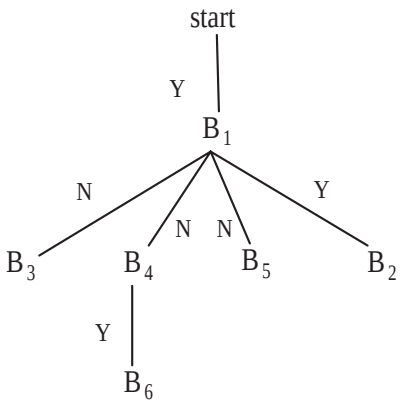


Figura 4: CDG Básico para o Grafo de Fluxo da Figura 2

6 Dependências entre Objetos Alocados Dinamicamente

Um outro tipo de dependência cuja determinação pode ser importante em otimizações é aquela existente em estruturas de dados armazenadas dinamicamente, como listas, árvores, grafos etc. Por exemplo, se determinamos que uma determinada estrutura de dados é uma lista encadeada, este fato pode ser usado para melhorar a alocação de dados em um *cache*, a exemplo do que é feito com vetores.

Uma das técnicas descritas brevemente em [2] divide-se em três partes: (1) proposição de um esquema de nomes para as localizações do *heap*; (2) estabelecimento de um conjunto de axiomas descrevendo as propriedades básicas das estruturas de dados sob análise; e (3) uso de um provador de teoremas para determinar propriedades desejáveis na estrutura de dados (por exemplo, se a estrutura de dados é uma fila então itens são adicionados em uma extremidade e removidos da outra).

No entanto, todas as técnicas até hoje propostas para análise de estruturas de dados dinâmicas

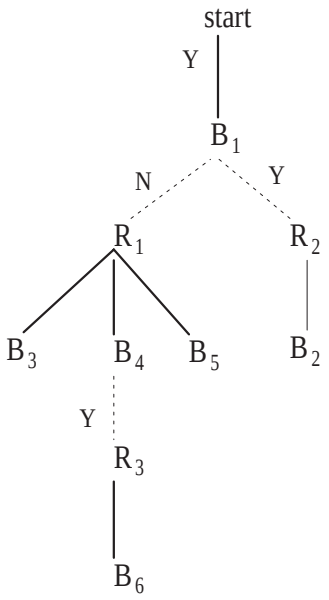


Figura 5: CDG com adição de nós de região para o Grafo de Fluxo da Figura 2

têm como desvantagem o grande esforço computacional que requerem, o que torna improvável o seu efetivo uso em compiladores comerciais.

7 Conclusões

Neste trabalho, descreveu-se as principais técnicas de análise de dependência, baseando-se em informações contidas em um capítulo de mesmo nome existente em [2]. Algumas outras informações foram obtidas em [1].

Análise de dependência procura identificar as relações de precedência entre as instruções de um programa que devem ser obedecidas para uma correta execução do mesmo. É, portanto, uma ferramenta importante para escalonamento de instruções e para as otimizações de *caches* de dados. Descreveu-se no trabalho os tipos de precedência que existem e como elas podem ser representadas em um Grafo de Dependência. Mostrou-se ainda um algoritmo simples para construção deste grafo.

Descreveu-se as dependências que podem existir entre variáveis indexadas utilizadas no interior de *loops*. Em seguida, mostrou-se uma forma recente de representação intermediária, conhecida por Grafo de Dependência de Programas (CDG). Por fim, discutiu-se sucintamente sobre as dependências que podem existir entre objetos alocados dinamicamente.

Referências

[1] Hennessy, J.L. and Patterson, D.A. *Computer architecture a quantitative approach*. Second Edition, Morgan Kaufmann Publishers, 1995.

[2] Muchnick, S. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

4 Análise de Dependência

Análise de Alias

Vladmir Oliveira Di Iorio

Análise de Alias

Vladimir Oliveira Di Iorio

1 Introdução

Análise de *alias* é a determinação de posições de armazenamento que podem ser acessadas de duas ou mais maneiras dentro de um programa. Por exemplo, uma variável C pode ser acessada por meio de seu nome ou por um apontador que contenha seu endereço, como mostrado na Figura 1. Determinar os possíveis *aliases* dentro de um programa é fundamental para executar otimizações corretamente, enquanto que a minimização dos conjuntos de *aliases* encontrados é essencial para que as otimizações sejam tão agressivas quanto possível.

Observe o exemplo da Figura 2. A segunda atribuição `k = a + 5` é redundante se e somente se a chamada a `f()` e a atribuição por meio do apontador q não modificarem o valor de `a`. Este exemplo mostra a importância da determinação de *alias* tanto dentro de procedimentos quanto entre eles.

É interessante distinguir *aliases* quanto à certeza de ocorrer em qualquer ponto do programa:

- Alias *possível* (*may* alias): indica *alias* que pode ocorrer.
- Alias *absoluto* (*must* alias): indica *alias* que ocorre com certeza.

Por exemplo, se todo caminho em um procedimento inclui uma atribuição do endereço de uma variável `x` a uma variável apontador `p` e esse é o único valor atribuído a `p`, então "`p` aponta para `x`" é uma informação de *alias* absoluto. Por outro lado, se em alguns caminhos uma variável ponteiro `q` recebe o endereço de `y` e em outros caminhos recebe o endereço de `z`, então "`q` pode apontar para `y` ou `z`" é uma informação de *alias* possível.

A classificação das informações sobre *alias* em prováveis ou absolutas é muito importante para otimizações. Se indica que uma propriedade **é** válida, então pode ser tomada como premissa de uma otimização. Se indica que uma propriedade

```
void main(){
  int *p;
  int n;
  p = &n;
  n = 4;
  printf (''%d\n'', *p);
}
```

Figura 1: Alias de Apontador em C.

```
...
int a, k;
extern int *q;
...
k = a + 5;
f (&a);
*q = 13;
k = a + 5; /* redundante?  */
...
```

Figura 2: Exemplo de Importância da Identificação de *Aliases*.

pode ser válida, então deve ser avaliada em termos de otimização, mas não pode ser contada como certa.

Outra informação útil é a distinção entre:

- informação de *alias sensível ao fluxo* e
- informação de *alias não-sensível ao fluxo*.

Um exemplo de informação não-sensível ao fluxo é "`p` pode apontar para `x` porque existe um caminho no qual `p` recebe o endereço de `x`". Essa afirmação indica uma relação de *alias* que pode valer em algum ponto do procedimento. Um exemplo sensível ao fluxo seria "`p` aponta para `x` no bloco B7".

Problemas não-sensíveis ao fluxo podem geralmente ser resolvidos por meio da resolução de subproblemas, seguido da combinação das soluções para resolver o problema completo. Problemas sensíveis ao fluxo, por outro lado, requerem que cada caminho do grafo de fluxo de controle seja seguido para computar a solução.

A formalização utilizada para representar *aliases* vai depender de que tipo de informação é necessária. São quatro as possibilidades: *alias* provável sensível ao fluxo, *alias* provável não-sensível ao fluxo, *alias* absoluto sensível ao fluxo, *alias* absoluto não-sensível ao fluxo. Assim, as informações sobre *alias* provável não-sensível ao fluxo podem ser representadas por uma relação binária $alias \in Var \times Var$. Nesse caso, x *alias* y se e somente se x e y podem, posivelmente em momentos diferentes, referenciar a mesma área de armazenamento. Por outro lado, informações sobre *alias* provável sensível ao fluxo devem envolver pontos de programa diferentes. Uma forma de representação seria uma função que mapeia pontos de programa e variáveis em conjutos de áreas de armazenamento abstratas. Esta última abordagem será a adotada nas seções seguintes.

As fontes de *alias* variam de linguagem para linguagem , mas existem características comuns a qualquer linguagem. Por exemplo, uma linguagem pode permitir que duas variáveis se sobreponham, isto é, indiquem o mesmo objeto, ou pode permitir que uma variável aponte para outra. Independente das regras específicas a cada linguagem, se uma variável `a` é apontada por `b` e `b` é apontada por `c` em um ponto da execução, então `a` é alcançável por meio de `c`. Para separar os aspectos específicos de cada linguagem, a computação dos *alias* é dividida em duas partes:

1. *Obtenção de* alias *(*alias *gatherer)*: componente específico de cada linguagem, fornecido pelo *front end* do compilador.

2. *Propagação de* alias *(*alias *propagator)*: componente do otimizador independente da linguagem. Realiza análise de fluxo de dados usando as relações de *alias* fornecidas pelo *front end* e transferindo as informações para os pontos onde são necessárias.

A obtenção de *alias* específico de cada linguagem pode descobrir *alias* existentes devido a:

- sobreposição da memória alocada para dois objetos;
- referência a arranjos;
- referências por meio de apontadores;
- passagem de parâmetros; ou
- combinação dos mecanismos acima.

2 *Aliases* em Linguagens de Programação

Nesta seção iremos discutir as diferentes formas que possibilitam a criação de *alias* nas linguagens Pascal e C.

2.1 *Aliases* em Pascal

Em Pascal padrão ANSI, existem diversos mecanismos que permitem a criação de *aliases*. Dentre eles, podemos mencionar:

1. Registros em Pascal podem possuir variantes que podem ser rotulados ou não. Se uma variável é um registro variante não rotulado, seus campos variantes podem ser acessados por dois ou mais conjuntos de nomes.

2. Uma variável do tipo apontador com um valor válido só pode conter `nil` ou apontar para objetos de um tipo específico. A linguagem não permite a obtenção do endereço de uma variável, assim um apontador não-nulo só pode indicar um objeto alocado dinamicamente pelo procedimento `new()`. Vários apontadores podem indicar o mesmo objeto em um determinado momento da execução, configurando uma situação de *alias*.

3. Pascal possui passagem de parâmetros por referência, o que permite que uma rotina modifique o valor do argumento real associado ao parâmetro, criando um *alias*.

4. Definições de procedimentos podem ser aninhadas. Assim, um procedimento pode acessar uma mesma variável por meio de seu nome ou por meio de um parâmetro de referência.

5. Uma função pode retornar um apontador, criando *aliases* para objetos alocados dinamicamente.

```
int a[100], *p, *q;
...
p = a; q = &a[0];
...
a[2] = 1;
p[2] = 1;
*(q + 2) = 1 ;
```

Figura 3: Arrays e apontadores em C.

```
void f (int *i, int *j) {
  *i = *j + 1;
}
...
f(&k, &k);
```

Figura 4: Alias em Passagem de Parâmetros em C.

2.2 *Aliases* em C

A linguagem C possui uma complexa variedade de mecanismos de criação de *aliases*. Dentre os mais importantes, podemos mencionar:

1. *Aliases* estáticos podem ser criados por meio do especificador de tipo `union`. Um tipo `union` pode ter vários campos declarados, todos com armazenamento sobreposto.

2. C permite alocação dinâmica de objetos, que podem ser referenciados simultaneamente por vários apontadores. Assim apontadores podem ser *alias* de outros. Além disso, é possível computar o endereço de um objeto com o operador &, independente deste ter alocação estática, automática ou dinâmica. O acesso pode ser realizado por meio do apontador que contém o endereço do objeto.

3. C permite ainda realizar aritmética de apontadores e os considera equivalentes a uma indexação de array. Por exemplo, na Figura 3, todos os comandos de atribuição exibidos são equivalentes. Aritmética de apontadores poderia ser utilizada de maneira indiscriminada para percorrer qualquer ponto da memória e criar *aliases* arbitrários, mas o padrão ANSI determina que o comportamento de um código como esse é indefinido.

4. Embora toda passagem de parâmetro seja por valor, *aliases* podem ser criados em chamadas de funções porque os parâmetros podem ser apontadores para qualquer objeto. Além disso, não há restrição quanto a passar o mesmo objeto como dois argumentos distintos para uma função, como na Figura 4.

5. Aninhamento de procedimentos não é permitido, mas procedimentos podem acessar variáveis globais. Um objeto global pode então ser referenciado pelo nome ou por um apontador dentro de um procedimento.

6. Como em Pascal, uma função pode retornar um apontador, criando *aliases* para objetos alocados dinamicamente.

3 Obtenção de Alias

Nesta seção, utilizaremos várias relações que representam possíveis *aliases* entre objetos de uma linguagem de programação. Como explicado anteriormente, esta é uma fase específica a cada linguagem, que depende de informações fornecidas pelo *front end*. Vamos concentrar os exemplos na linguagem C, que possui mecanismos bem complexos de formação de *aliases*.

No desenvolvimento a seguir, uma série de decisões arbitrárias deverão ser tomadas. Por exemplo, em C, se tivermos uma estrutura `s` com campos `s1` e `s2`, então com certeza existe uma interseção entre a área de armazenamento de `s` e a área de `s.s1` e `s.s2`. Entretanto, vamos arbitrar que não existe nenhuma sobreposição entre as áreas de `s.s1` e `s.s2`. Esta distinção requer gasto maior de espaço, mas pode gerar um código melhor. A sobreposição entre `s.s1` e `s.s2` ocorreria se assumíssemos um não-alinhamento dos campos de `s`.

Outra decisão importante é que não vamos tentar fazer uma distinção entre cada uma das áreas de armazenamento alocadas dinamicamente. Vamos tratá-las como uma única área, com exceção da distinção usando tipo que pode ser utilizada em Pascal.

Para a obtenção de *aliases*, vamos considerar que um procedimento é representado por grafo de fluxo onde os vértices representam comandos individuais e as arestas representam o fluxo de controle. Outra alternativa seria considerar que os vértices são os blocos básicos.

3.1 Funções Básicas

Seja P um ponto do programa, i.e., um ponto entre dois comandos dentro do programa. No grafo de fluxo, um ponto de programa rotula uma aresta. Vamos supor que existem dois pontos especiais `entry+` e `exit-`, imediatamente antes do nodo de entrada do grafo e imediatamente após o nodo de saída, respectivamente. Seja $stmt(P)$ o único comando que imediatamente precede P no grafo de fluxo.

O grafo de fluxo da Figura 5 foi criado para ilustrar alguns dos conceitos usados nesta e nas seções seguintes. Uma única instrução é representada em cada nodo, e as arestas são rotuladas com os pontos do programa. Temos, por exemplo, $stmt(1) =$ `n = 0` e $stmt($`exit-`$) =$ `return n`. O programa que esse grafo de fluxo representa pode ser visto na Figura 6.

Seja x uma variável escalar, um arranjo, uma estrutura, um valor de um apontador etc. Então $mem_P(x)$ denota o espaço de memória associado com o objeto x, no ponto P de um programa. Seja $star(o)$ a área de memória alocada estática ou dinamicamente, ocupada pelo objeto sintático o. Seja $anon(ty)$ a área "anônima" de memória dinâmica alocada para objetos do tipo ty e seja $anon$ toda a memória dinâmica alocada em um procedimento.

Para todos P e x, $mem_P(x)$ é:

- $star(x)$ se x é estática ou automaticamente alocado,

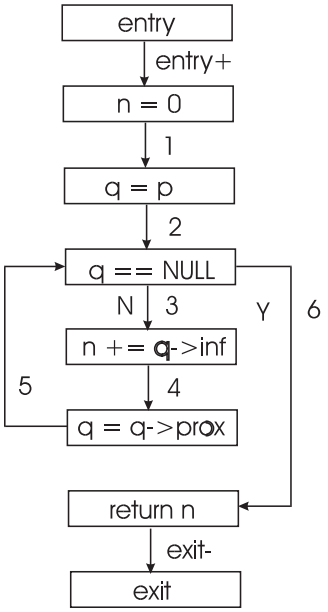


Figura 5: Grafo de Fluxo de um Programa C.

```
typedef struct {node *prox; int inf} node;

int ex1 (node *p) {
  int n = 0;
  for (node *q = p; q == NULL; q = q->prox)
    n += q->inf;
  return n;
}
```

Figura 6: Programa Representado pelo Grafo da Figura 5.

- $anon(ty)$ se x é dinamicamente alocado (onde ty é o tipo de x), ou
- $anon$ se x é dinamicamente alocado e seu tipo não é conhecido.

Na Figura 5, a memória associada com `n` no ponto 1, denotada por mem_1(`n`), é $star$(`n`). A memória associada com `*p` no ponto 3, denotada por mem_3(`*p`), é $anon$(`node`).

Seja any o conjunto de todas as possíveis áreas de armazenamento e $any(ty)$, onde ty é um tipo, todas as possíveis áreas do tipo ty. Seja ainda $globals$ o conjunto de todas as possíveis áreas de armazenamento acessíveis globalmente. As funções $anon(ty)$ e $any(ty)$ são úteis para Pascal, uma vez que apontadores só podem apontar para áreas de um tipo fixo determinado.

Vamos definir uma série de funções que mapeiam pontos de programa P e objetos x que podem armazenar valores, i.e., variáveis, arrays, campos de registros etc, em áreas de armazenamento abstratas. Quando o objeto for necessariamente um apontador, vamos denotá-lo por p:

1. $ovr_P(x)$ = o conjunto de áreas de armazenamento abstratas que podem ter interseção com a área de x no ponto de programa P.
2. $ptr_P(p)$ = o conjunto de áreas de armazenamento abstratas que p pode apontar no ponto de programa P.
3. $ref_P(p)$ = o conjunto de áreas de armazenamento abstratas alcançáveis por meio do apontador p, no ponto de programa P, seguindo uma cadeia arbitrariamente grande de apontadores.
4. $ref(p)$ = o conjunto de áreas de memória abstrata alcançáveis por meio do apontador p, independente do ponto de programa, seguindo uma cadeia arbitrariamente grande de apontadores.

Na Seção 3.2 a seguir, vamos exibir as regras que permitem a criação de *aliases* em C, utilizando as funções definidas acima.

3.2 Regras para *aliases* na linguagem C

As regras desta seção não descrevem completamente todas os possíveis casos de *aliases* em C, mas são suficientes para exemplificar a sua formalização.

Nas regras exibidas a seguir, por simplicidade, um apontador para um elemento de um arranjo irá ser tratado como um *alias* para o arranjo inteiro. Nestas regras e nas seções seguintes, P é um ponto de programa e P' é o, geralmente único, ponto de programa que precede P. Tratamento especial será dado aos casos de múltiplos pontos precedentes.

1. Se $stmt(P)$ atribui um apontador nulo a p, então

$$ptr_P(p) = \emptyset$$

2. Se $stmt(P)$ atribui uma área de armazenamento alocada dinamicamente a p, por exemplo, por uma chamada a `malloc( )`, então

$$ptr_P(p) = anon$$

3. Se $stmt(P)$ é "p `=` `&`a", então

$$ptr_P(p) = \{mem_P(a)\} = \{mem_{P'}(a)\}$$

4. Se $stmt(P)$ é "$p1$ `=` $p2$", então

$$ptr_P(p1) = ptr_P(p2) = \begin{cases} mem_{\texttt{entry+}}(*p2) & \text{se } P' = \texttt{entry+} \\ ptr_{P'}(p2) & \text{caso contrário} \end{cases}$$

5. Se $stmt(P)$ é "$p1$ `=` $p2$`->`$p3$", onde $p3$ é um campo apontador, então

$$ptr_P(p1) = ptr_{P'}(p2\texttt{->}p3)$$

6. Se $stmt(P)$ é "$p1$ `=` `&`a`[`$expr$`]`", onde a é um arranjo, então

$$ptr_P(p) = ovr_P(a) = ovr_{P'}(a) = \{mem_{P'}(a)\}$$

7. Se $stmt(P)$ é "p `=` p `+` i", onde i é um valor inteiro, então

$$ptr_P(p) = ptr_{P'}(p)$$

8. Se $stmt(P)$ é "`*`p `=` a", então

$$ptr_P(p) = ptr_{P'}(p)$$

e se `*`p for um apontador, então

$$ptr_P(*p) = ptr_P(a) = ptr_{P'}(a)$$

9. Se $stmt(P)$ testa "p `==` q" e P rotula o ponto de programa associado à avaliação da condição como verdadeira, então

$$ptr_P(p) = ptr_P(q) = ptr_{P'}(p) \cap ptr_{P'}(q)$$

uma vez que, sendo a condição verdadeira, p e q apontarão para a mesma área[1].

10. Se st denota um tipo `struct` com campos s_1 até s_n, e s é um objeto estático ou automático do tipo st, então

$$ovr_P(s) = \{mem_P(s)\} = \bigcup_{i=1}^{n} \{mem_P(s.s_i)\}$$

e também, para cada i,

$$\{mem_P(s.s_i)\} = ovr_P(s.s_i) \subset ovr_P(s)$$

e para todo $j \neq i$,

$$ovr_P(s.s_i) \cap ovr_P(s.s_j) = \emptyset$$

[1]Nenhuma informação nova está disponível no ponto de programa associado à avaliação da condição como falsa, a não ser que uma análise bem mais complexa seja conduzida.

11. Se st denota um tipo `struct` com campos s_1 até s_n, e p é um apontador tal que $stmt(P)$ aloca um objeto s do tipo st, então

$$ptr_P(p) = \{mem_P(*p)\} = \bigcup_{i=1}^{n} \{mem_P(p\text{->}s_i)\}$$

e para cada i,

$$\{mem_P(*p\text{->}s_i)\} = ptr_P(p\text{->}s_i) \subset ptr_P(s)$$

e para todo $j \neq i$,

$$ptr_P(p\text{->}s_i) \cap ptr_P(p\text{->}s_j) = \emptyset$$

e para todos objetos x,

$$ptr_P(p) \cap \{mem_P(x)\} = \emptyset$$

uma vez que, após a alocação do objeto, cada campo tem um endereço distinto.

12. Se ut denota um tipo `union` com componentes u_1 até u_n, e u é um objeto estático ou automático do tipo ut, então, para $i = 1, \ldots, n$,

$$ovr_P(u) = \{mem_P(u)\} = \{mem_P(u.u_i)\}$$

13. Se ut denota um tipo `union` com componentes u_1 até u_n, e p é um apontador tal que $stmt(P)$ aloca um objeto do tipo ut, então, para $i = 1, \ldots, n$,

$$ptr_P(p) = \{mem_P(*p)\} = \{mem_P(p\text{->}u_i)\}$$

e para todos os outros objetos x,

$$ptr_P(p) \cap \{mem_P(x)\} = \emptyset$$

14. Se $stmt(P)$ inclui uma chamada para uma função $f(\ )$, então

$$ptr_P(p) = ref_{P'}(p)$$

para todos os apontadores p que são argumentos de $f(\ )$, para aqueles que são globais, ou para aqueles que recebem um valor retornado por $f(\ )$.

4 Propagação de Alias

Agora que temos um método para definir as fontes de *aliases* de uma maneira independente de linguagem, vamos descrever o componente que executa a propagação dessas informações.

4.1 Funções de Fluxo

Para descrever o propagador de *aliases*, vamos utilizar uma análise de fluxo de dados baseada nas funções $ovr_p(\ )$ e $ptr_p(\ )$ da Seção 3.1. As funções globais de fluxo associadas terão o mesmo nome, com inicial maiúscula.

Seja $\mathbf{P}$ o conjunto de pontos de programa em um procedimento, $\mathbf{O}$ o conjunto de objetos visíveis nos mesmos, e $\mathbf{S}$ o conjunto de áreas de memória abstrata. Então $Ovr : \mathbf{P} \times \mathbf{O} \rightarrow 2^{\mathbf{S}}$ mapeia pontos de programa e objetos em conjuntos de áreas de alocação de memória abstratas que podem se sobrepor e $Ptr : \mathbf{P} \times \mathbf{O} \rightarrow 2^{\mathbf{S}}$ mapeia pontos de programa e objetos, que serão sempre apontadores, em conjuntos de áreas de alocação de memória abstratas apontadas pelos objetos. Sua definição é a seguinte:

1. Seja P um ponto de programa tal que $stmt(P)$ tenha um único predecessor P'. Então

$$Ovr(P,x) = \begin{cases} ovr_P(x) & \text{se } stmt(P) \text{ afeta } x \\ Ovr(P',x) & \text{caso contrário} \end{cases}$$

   e

$$Ptr(P,p) = \begin{cases} ptr_P(p) & \text{se } stmt(P) \text{ afeta } p \\ Ptr(P',p) & \text{caso contrário} \end{cases}$$

2. Suponha que $stmt(P)$ tenha múltiplos predecessores P_1 até P_n e assuma, por simplicidade, que $stmt(P)$ seja o comando vazio. Então, para qualquer objeto x,

$$Ovr(P,x) = \bigcup_{i=1}^{n} Ovr(P_i,x)$$

   e para qualquer variável apontador p,

$$Ptr(P,p) = \bigcup_{i=1}^{n} Ptr(P_i,p)$$

3. Seja P um ponto de programa seguido por um teste (que vamos assumir, por simplicidade, que não chama nenhuma função nem modifica nenhuma variável) com múltiplos pontos sucessores P_1 até P_n. Então, para cada i e qualquer objeto x,

$$Ovr(P_i,x) = Ovr(P,x)$$

   e para qualquer variável apontador p,

$$Ptr(P_i,p) = Ptr(P,p)$$

   exceto que se distinguirmos o resultado positivo de um teste, podemos obter uma informação mais precisa.

```
int ex2 (int n){
  int i, j, k, *p, *q;
  p = &i;
  i = n + 1;
  q = &j;
  j = n * 2;
  k = *p + *q;
  return k;
}
```

Figura 7: Exemplo de Análise de *alias* em C.

Como valores iniciais para as funções $Ovr_p(\ )$ e $Ptr_p(\ )$ usaremos, para objetos locais x:

$$Ovr(P,x) = \begin{cases} \{star(x)\} & \text{se } P = \texttt{entry+} \\ \emptyset & \text{caso contrário} \end{cases}$$

e para apontadores p:

$$Ptr(P,p) = \begin{cases} \emptyset & \text{se } P = \texttt{entry+} \text{ e } p \text{ é local} \\ any & \text{se } P = \texttt{entry+} \text{ e } p \text{ é global} \\ \{mem_{\texttt{entry+}}(*p)\} & \text{se } P = \texttt{entry+} \text{ e } p \text{ é um parâmetro} \\ \emptyset & \text{caso contrário} \end{cases}$$

onde $star(x)$ denota a área de memória alocada para x para satisfazer sua declaração local.

4.2 Exemplos

Considere o código C exibido na Figura 7 e o grafo de fluxo correspondente na Figura 8.

A função $Ovr(\ )$ possui apenas valores triviais, assim vamos nos concentrar na função $Ptr(\ )$:

$$\begin{array}{lcl} Ptr(\texttt{entry+},\texttt{p}) & = & \emptyset \\ Ptr(\texttt{entry+},\texttt{q}) & = & \emptyset \\ Ptr(1,\texttt{p}) & = & ptr_1(\texttt{p}) \\ Ptr(3,\texttt{q}) & = & ptr_3(\texttt{q}) \end{array}$$

e $Ptr(P,x) = Ptr(P',x)$ para todos os outros pares de pontos de programa P e apontadores x. A solução para as equações é facilmente computada por substituições:

$$\begin{array}{lcllcl} Ptr(\texttt{entry+},\texttt{p}) & = & \emptyset & Ptr(\texttt{entry+},\texttt{q}) & = & \emptyset \\ Ptr(1,\texttt{p}) & = & \{star(\texttt{i})\} & Ptr(1,\texttt{q}) & = & \emptyset \\ Ptr(2,\texttt{p}) & = & \{star(\texttt{i})\} & Ptr(2,\texttt{q}) & = & \emptyset \\ Ptr(3,\texttt{p}) & = & \{star(\texttt{i})\} & Ptr(3,\texttt{q}) & = & \{star(\texttt{j})\} \\ Ptr(4,\texttt{p}) & = & \{star(\texttt{i})\} & Ptr(4,\texttt{q}) & = & \{star(\texttt{j})\} \\ Ptr(5,\texttt{p}) & = & \{star(\texttt{i})\} & Ptr(5,\texttt{q}) & = & \{star(\texttt{j})\} \\ Ptr(\texttt{exit-},\texttt{p}) & = & \{star(\texttt{i})\} & Ptr(\texttt{exit-},\texttt{q}) & = & \{star(\texttt{j})\} \end{array}$$

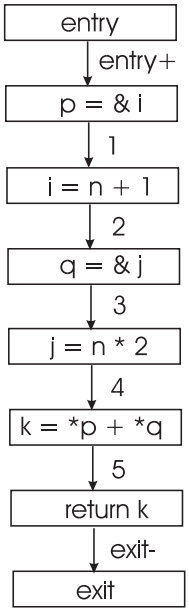


Figura 8: Grafo de Fluxo para o Programa da Figura 7.

O fato de que os valores apontados por p e q são apenas consultados e nunca modificados nos mostra que não há nenhum problema criado por *alias* nesta rotina. Isso permite que possamos substituir `k = *p + *q` por `k = i + j` e remover as atribuições para p e q completamente.

Como um segundo exemplo, considere o grafo da Figura 5, na Seção 3.1. As equações para $Ptr(\ )$ são as seguintes:

$$\begin{array}{lcl} Ptr(\texttt{entry+}, \texttt{p}) & = & \{mem_{\texttt{entry+}}(\texttt{*p})\} \\ Ptr(2, \texttt{q}) & = & ptr_2(\texttt{q}) \\ Ptr(3, \texttt{q}) & = & Ptr(2, \texttt{q}) \cup Ptr(5, \texttt{q}) \\ Ptr(5, \texttt{q}) & = & ptr(5, \texttt{q}) \\ Ptr(6, \texttt{q}) & = & ptr(6, \texttt{q}) \\ Ptr(\texttt{exit-}, \texttt{q}) & = & Ptr(6, \texttt{q}) \end{array}$$

Para resolver as equações, faremos uma série de substituições, resultando em:

$$\begin{array}{lcl} Ptr(\texttt{entry+}, \texttt{p}) & = & \{mem_{\texttt{entry+}}(\texttt{*p})\} \\ Ptr(2, \texttt{q}) & = & ptr_1(\texttt{p}) = Ptr(\texttt{entry+}, \texttt{p}) = \{mem_{\texttt{entry+}}(\texttt{*p})\} \\ Ptr(3, \texttt{q}) & = & \{mem_{\texttt{entry+}}(\texttt{*p})\} \cup Ptr(5, \texttt{q}) \\ Ptr(5, \texttt{q}) & = & ptr(4, \texttt{q->prox}) = ptr(3, \texttt{q->prox}) \\ Ptr(6, \texttt{q}) & = & \{\texttt{NULL}\} \\ Ptr(\texttt{exit-}, \texttt{q}) & = & \{\texttt{NULL}\} \end{array}$$

Outras rodadas de substituições revelam as informações seguintes, sendo que as demais não sofrem alterações:

$$\begin{array}{lcl} Ptr(3, \texttt{q}) & = & \{mem_{\texttt{entry+}}(\texttt{*p})\} \cup ref_3(\texttt{q}) \\ Ptr(5, \texttt{q}) & = & \{mem_{\texttt{entry+}}(\texttt{*p})\} \cup ref_3(\texttt{q}) \end{array}$$

e pode-se facilmente ver que o valor de $ref_3(\texttt{q})$ é equivalente a $ref_{\texttt{entry+}}(\texttt{p})$. Ou seja, dentro do loop, a variável `q` pode ser um *alias* para qualquer valor acessível a partir

de `p` na entrada da rotina, mas nenhum outro. Note que isso inclui apenas valores alocados **fora** da rotina.

5 Conclusão

As informações sobre análise de *alias* exibidas no texto foram obtidas, na sua maior parte, do livro sobre implementação de compiladores de Muchnick [2]. Alguns exemplos foram baseados também no texto de Appel [1].

A análise de *alias* é essencial para otimizações mais agressivas. Isso acontece porque devemos saber, com certeza, que foram levadas em conta todas as maneiras pelas quais uma determinada variável ou posição de memória poderá ser utilizada ou modificada.

Se falharmos em identificar uma possibilidade de *alias*, nós podemos errar em duas direções diferentes. Podemos errar modificando a semântica do programa ou por não executar uma otimização que seria possível. Embora ambas sejam indesejáveis, vamos obviamente sempre ser conservadores e escolher a segunda, quando não for possível inferir informações mais específicas sobre *alias*.

Embora informações de alta qualidade sobre *alias* sejam essenciais para otimizações agressivas, existe um grande número de programas para os quais uma pequena quantidade de informação é suficiente. Por exemplo, para a maioria dos programas escritos em C, é suficiente assumir que somente variáveis cujo endereço é computado são passíveis de *alias* e que qualquer ponteiro pode apontar para qualquer delas. Na maioria dos casos, essa consideração impõe restrições mínimas sobre a otimização.

Referências

1. Appel, A.W. *Modern Compiler Implementation in Java – Basic Techniques.* pp. 351–356. Cambridge University Press. Cambridge, UK. 1997.

2. Muchnick, S.S. *Advanced Compiler Design and Implementation.* pp. 293-317. Morgan Kaufmann Publishers. San Francisco, USA. 1997.

5 Análise de Alias

Avaliação Parcial de Programas

Vladmir Oliveira Di Iorio

Avaliação Parcial de Programas

Vladimir Oliveira Di Iorio

Junho de 1999

Índice

Capítulo 1

Introdução

Este documento apresenta conceitos relacionados à avaliação parcial de programas. Contém definições, exemplos, aplicações e tópicos de pesquisa. As referências principais para a construção do texto foram o livro de Jones, Gomard e Sestoft [22], um relatório de Mogensen e Sestoft [32], e o material produzido por John Hatcliff para a Escola de Verão do DIKU de 1998 (*Partial Evaluation: Practice and Theory*). O Capítulo 5 discute essas fontes de referência e mostra como obtê-las.

A Seção 1.1 a seguir apresenta uma definição informal de avaliação parcial de programas e um exemplo simples. Resume os principais assuntos discutidos neste documento, exibindo sua distribuição nas demais seções. Pode servir como guia para a leitura do resto do documento.

1.1 Avaliação Parcial de Programas

Suponha um programa P que tenha duas entradas, identificadas como in_1 e in_2. O resultado da *avaliação parcial* de P com relação à entrada in_1 é um novo programa P_{in_1}, designado por *residual* ou *especializado* [22]. O programa P_{in_1}, quando executado sobre a entrada restante in_2, produz o mesmo resultado que a execução de P sobre ambas as entradas. Avaliação parcial é um tipo de *especialização de programas*. A entrada in_1 é designada *entrada estática*, e in_2, *entrada dinâmica*.

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

Um *avaliador parcial* para uma linguagem L é um programa que executa avaliação parcial sobre programas escritos em L, produzindo como resultado programas especializados que são escritos, geralmente, na mesma linguagem L. O avaliador parcial realiza uma mistura de execução com geração de código,

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

Figura 1.1: Função Power(n,x) = x^n.

motivo pelo qual o processo foi designado "*mixed computation*", e o avaliador comumente chamado de `mix` [14, 24].

Observe a função *Power(n,x)* na Figura 1.1, escrita na linguagem C, que computa o valor de x^n. A especialização de *Power* dado $n = 5$ é uma função com apenas um parâmetro x. Essa função deve produzir o mesmo resultado que *Power*, se for executada com entradas 5 e x, para qualquer valor de x.

Uma solução ingênua, mas que satisfaz a condição imposta acima, seria:

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

De fato, o *Teorema s-m-n* de Kleene [22] mostra que sempre é possível obter um programa especializado, e sua prova exibe o projeto de um avaliador parcial. Esse teorema, entretanto, não se preocupa com questões de eficiência.

No exemplo em questão, o uso do valor estático n permite obter um código mais eficiente. Um avaliador parcial para a linguagem C deveria produzir um programa residual como o exibido a seguir, ao especializar a função Power com respeito a $n = 5$:

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

Avaliadores parciais já foram desenvolvidos com sucesso para linguagens de paradigma imperativo [2, 3, 28], funcional [21, 25, 7, 30] e lógico [17, 36, 31].

A avaliação parcial de programas é utilizada em diversas áreas da computação. Para que sua utilização valha a pena, deve-se pesar se o custo de se produzir um programa especializado, com relação a um determinado parâmetro de entrada, é compensado pelo ganho de velocidade na execução. Se uma das entradas varia com freqüência bem menor que as outras, o custo de produção do programa especializado é diluído em um conjunto de execuções. Esse é o caso mais indicado para se utilizar a avaliação parcial. Entretanto, muitas outras situações podem se valer das técnicas apresentadas neste capítulo. Veremos isso na Seção 1.2, que apresenta diversas aplicações de avaliação parcial.

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

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

No Capítulo 2, apresentamos a especialização polivariante e discutimos as diferenças e vantagens das duas abordagens (*online* e *offline*). Uma linguagem simples de fluxograma, designada FCL, é definida e é utlizada nos exemplos da seção inteira.

O Capítulo 3 traz alguns exemplos de avaliação parcial de programas, usando as técnicas introduzidas no Capítulo 2. Todos os exemplos são escritos na linguagem FCL e a especialização é realizada usando a abordagem *offline*.

A compilação e geração de compiladores é uma das aplicações mais importantes de avaliação parcial, assim é discutida separadamente no Capítulo 4. Uma definição mais formal de avaliação parcial é exibida, servindo de base para a apresentação das *Três Projeções de Futamura*. Essas projeções são equações que mostram como, usando especialização de interpretadores, é possível realizar compilação dirigida por semântica; a auto-aplicação do avaliador parcial permite ainda geração de compiladores e geração de geradores de compiladores. Finalmente, é apresentada uma abordagem diferente para a avaliação parcial, designada *geração de extensões*. São apresentados argumentos que defendem a utilização dessa segunda abordagem quando se pretende realizar geração de compiladores para linguagens fortemente tipadas.

O Capítulo 5 enumera algumas fontes de referência adicionais sobre avaliação parcial de programas.

1.2 Aplicações de Avaliação Parcial

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

1.2.1 Engenharia de Software

Dois objetivos importantes na produção de programas são geralmente conflitantes:

- construir programas genéricos e modulares, facilitando o reuso de código e manutenção do mesmo;
- construir programas o mais eficiente possível.

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

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

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

1.2.2 Parâmetros com Freqüências de Variação Diferentes

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

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

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

O algoritmo geral recebe duas entradas, uma cena e um raio de luz. Na exibição de uma figura, a cena, que é uma coleção de objetos de três dimensões, não é alterada enquanto a seqüência de raios é traçada. A avaliação parcial do algoritmo de *ray tracing* com relação a uma cena específica resulta em um procedimento eficiente para traçar raios para aquela cena [5]. O tempo gasto para se construir o algoritmo especializado é compensado pela eficiência obtida, uma vez que o mesmo será utilizado inúmeras vezes para diferentes raios. Experiências realizadas por Mogensen mostraram um *ray tracer* especializado que era de 8 a 12 vezes mais rápido que o original [29].

1.2.3 Problemas com Natureza Interpretativa

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

Simuladores de circuitos recebem a descrição de um circuito elétrico como entrada, constroem equações diferenciais que descrevem seu comportamento e usam métodos numéricos para resolver essas equações. A especialização de um simulador geral com relação a um circuito específico gera um programa eficiente para esse circuito. Um ganho substancial em velocidade pode ser obtido [6].

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

Um algoritmo de casamento de padrões em textos recebe como entradas uma seqüência de caracteres definindo um padrão e um texto onde esse padrão deverá ser pesquisado. A avaliação parcial do algoritmo com relação a um padrão específico resulta em um programa que implementa um autômato finito determinístico que executa as ações necessárias para a identificação desse padrão em qualquer texto. Uma experiência interessante de Consel e Danvy mostra a geração de um algoritmo equivalente ao método de Knuth, Morris e Pratt (KMP) a partir de algoritmo de casamento de padrões ingênuo (força bruta) [11].

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

1.2.4 Compilação e Geração de Compiladores

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

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

Os conceitos envolvidos neste exemplo são um pouco mais complexos que os dos exemplos anteriores, assim serão explicados em detalhe na Seção 4.

Capítulo 2

Técnicas de Avaliação Parcial

Para apresentar as técnicas empregadas na avaliação parcial de programas, vamos utilizar uma linguagem de fluxograma designada FCL (*Flowchart Language*) [16]. Sendo bastante simples, essa linguagem é ideal para apresentarmos os conceitos básicos de avaliação parcial.

Vamos utilizar as seguintes notações:

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

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

2.1 Linguagem de Fluxograma FCL

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

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

Os comandos de atribuição têm o formato `v := exp`, onde `v` é uma variável e `exp` uma expressão contendo operações que envolvem variáveis e constantes. Um comando de desvio pode ser incondicional (`goto label`) ou condicional (`if boolexp goto label1 else label2`). O comando `return exp` retorna o valor da expressão calculada, terminando o programa. Todas as variáveis não pertencentes à entrada são inicializadas com zero. A Figura 2.1 mostra a codificação em FCL da função Power, exibida na Figura 1.1.

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

Figura 2.1: Codificação em FCL da função Power.

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

Figura 2.2: A execução da função Power com $n = 2$ e $x = 5$.

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

A execução de um programa FCL pode ser representada por uma seqüência de estados. Cada estado pode ser representado por um par (l, σ), onde l é um ponto do programa indicado por um rótulo e σ é o valor das variáveis antes da execução do comando indicado por l. A execução da função Power da Figura 2.1, com $n = 2$ e $x = 5$, é apresentada na Figura 2.2. Um novo rótulo, não existente no programa, foi introduzido para representar o término da execução e retorno de um valor: $\langle halt, 25\rangle$.

Na representação escolhida para execuções de FCL, o próximo comando a ser executado em um estado é sempre o primeiro comando de um bloco básico. Essa abordagem é adequada aos nossos propósitos de utilizar FCL para explicar o processo de avaliação parcial de programas, como veremos mais adiante.

Outra alternativa seria adotar uma granularidade mais fina, com o estado representando cada comando do programa.

2.2 Especialização Polivariante

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

Essa técnica é chamada de especialização polivariante porque um único ponto de programa do programa original pode dar origem a vários pontos de programa no programa especializado. Para entender o processo, considere como a execução exibida na Figura 2.2 teria que ser alterada se apenas parte dos dados de entrada fosse fornecida. Intuitivamente, podemos prever que alguns dos valores das variáveis representadas em cada estado seriam desconhecidos. Algumas das expressões, atribuições, e desvios poderiam ser computados, se dependessem apenas dos dados conhecidos, chamados de *entrada estática*. Outras estruturas do programa não poderiam ser computadas, se dependessem dos valores não conhecidos, chamados de *entrada dinâmica*. O processo procura estabelecer todos os estados alcançáveis, utilizando apenas os dados estáticos. Em seguida, constrói os blocos básicos do programa especializado. Cada bloco é derivado de um estado alcançado: o bloco construído a partir de (l, σ) é o resultado da computação de toda informação estática do bloco l, utilizando os valores conhecidos que aparecem em σ.

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

Resumindo, o processo é composto de três passos:

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

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

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

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

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

2.3 Métodos *Online*

Nas técnicas *online*, a avaliação parcial é geralmente executada em uma única fase, e os valores são definidos como estáticos ou dinâmicos durante a especialização. Nesta seção, vamos mostrar a abordagem *online* através de um exemplo, seguindo os três passos da especialização polivariante. Como dissemos antes, esses três passos são comumentes executadas em paralelo. Optamos por mostrar a execução de cada passo separadamente nos primeiros exemplos, por questões didáticas.

2.3.1 Determinação dos Estados Alcançáveis

Para demonstrar o processo, vamos voltar à execução do programa Power, exibida na Figura 2.2. Desta feita, vamos considerar apenas que o valor de n é 2 (entrada estática), mas o valor de x não é conhecido. O símbolo D vai ser utilizado para representar o valor de x (entrada dinâmica). Sendo assim, o estado inicial terá $b0$ como ponto de programa e $[n \mapsto 2, x \mapsto D, p \mapsto 0]$ como valor das variáveis. O valor inicial de p, embora não seja uma entrada conhecida, será 0, que é o valor *default* das variáveis. Apenas as entradas dinâmicas são inicializadas com D.

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

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

$$
\begin{array}{ll}
 & (b0, [n \mapsto 2, x \mapsto D, p \mapsto 0]) \\
\text{passo 1:} & (b1, [n \mapsto 2, x \mapsto D, p \mapsto 1]) \\
\text{passo 2:} & (b2, [n \mapsto 2, x \mapsto D, p \mapsto 1]) \\
\text{passo 3:} & (b3, [n \mapsto 2, x \mapsto D, p \mapsto 1]) \\
\text{passo 4:} & (b1, [n \mapsto 1, x \mapsto D, p \mapsto 1]) \\
\text{passo 5:} & (b2, [n \mapsto 1, x \mapsto D, p \mapsto 1]) \\
\text{passo 6:} & (b4, [n \mapsto 1, x \mapsto D, p \mapsto 1])
\end{array}
$$

passo 7: $(b1, [n \mapsto 0, x \mapsto D, p \mapsto D])$
passo 8: $(b5, [n \mapsto 0, x \mapsto D, p \mapsto D])$
passo 9: $(halt, [n \mapsto 0, x \mapsto D, p \mapsto D])$

O que acontece a cada passo:

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

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

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

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

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

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

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

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

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

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

- Se for gerado um estado que já foi produzido anteriormente na execução, o algoritmo de determinação dos estados alcançáveis não precisa repetir todo o processo. Esse caso irá corresponder a um *loop* no programa residual.

- No exemplo exibido, todos os testes dos desvios condicionais dependiam apenas dos valores estáticos, assim foi sempre possível identificar qual seria o próximo bloco a ser executado. Se a condição depender de valores dinâmicos, não teremos como computá-la. Nesse caso, o algoritmo deve gerar os estados alcançáveis levando em conta os dois blocos referenciados no desvio condicional.

2.3.2 Especialização dos Pontos de Programa

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

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

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

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

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

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

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

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

$(b3, [2, D, 1])$**:** Sempre que um dos valores de uma operação não for conhecido, a operação é classificada como D, ou seja, dinâmica. A variável x tem valor D, assim a expressão $x * x$ é classificada também como D. Portanto, `x :=`

`x*x;` deve aparecer no código residual. Por outro lado, na operação $n/2$, os valores envolvidos são uma variável estática e uma constante, assim a expressão pode ser calculada. E como n é estática, a atribuição `n := n/2;` vai ser computada, não aparecendo no código residual. Finalmente, um desvio especializado é produzido.

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

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

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

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

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

2.3.3 Compressão das Transições

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

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

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

Se aplicarmos a compressão de transições indiscriminadamente, podemos ter dois tipos de problemas: duplicação de código e compressão infinita. A duplicação de código ocorre quando a compressão é aplicada a duas transições

distintas para um mesmo ponto de programa. Quando o programa residual contém um *loop*, a compressão pode continuar indefinidamente.

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

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

A estratégia descrita ainda pode gerar duplicação de código, mas experiências relatadas indicam que é um problema mínimo [22]. O mais importante é que essa estratégia não produz uma compressão infinita, a não ser que o programa residual contenha um *loop* infinito que dependa apenas da entrada estática. Nesse caso, o programa original também entraria em *loop* infinito com os mesmos valores estáticos, independente dos valores dinâmicos utilizados.

Outras Otimizações

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

Usando Propriedades Algébricas A operação de multiplicação

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

poderia ser otimizada para

```
p := x;
```

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

Expandindo Expressões No código a seguir:

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

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

Figura 2.3: MIX - Algoritmo para Especialização de Programas.

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

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

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

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

poderia ser expandido para

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

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

2.3.4 Especialização Online: Três Passos em Paralelo

Um algoritmo para realizar especialização de programas FCL (que na Seção 1.1 chamamos de `mix`), executando os três passos em paralelo, é exibido na Figura 2.3. O algoritmo tem como entradas o programa original (`program`), o rótulo do bloco inicial do programa ($\texttt{pp}_0$) e o valor inicial das variáveis ($\texttt{vs}_0$).

O valor de cada variável é representado por um par $(type, val)$, onde $type$ é S (estático) ou D (dinâmico). Se a variável for dinâmica, val contém uma expressão representando a própria variável; se for estática, val contém a constante

associada. No exemplo desenvolvido nesta seção, a lista de valores iniciais $\texttt{vs}_0$ seria: $[n \mapsto (S, 2)\,,\ x \mapsto (D, x)\,,\ p \mapsto (S, 0)]$. Em métodos *online*, a utilização dos "*tags*" S e D para diferenciar valores estáticos e dinâmicos é essencial.

O conjunto `pending` contém os blocos do programa residual que ainda não foram gerados. Esses blocos são identificados por um par (pp, vs), onde pp é um rótulo do programa original e vs é uma lista de valores para as variáveis. O conjunto `marked` é utilizado para armazenar os rótulos dos blocos gerados no programa residual, para evitar o processamento de um bloco que já foi gerado.

O *loop* mais interno processa cada comando FCL do bloco corrente. Esses comandos podem alterar o estado vs das variáveis e também gerar novos blocos a serem inseridos em `pending`. O código para processar cada comando é exibido na Figura 2.4.

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

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

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

Na parte inicial do código exibido na Figura 2.4, pode-se ver o processamento dos comandos de atribuição e retorno de valor. Em um comando `return`, primeiro a expressão é processada. Se resultar em um valor estático, ela deve sofrer *lift* para aparecer no código residual, uma vez que todo comando `return` é residualizado. Em um comando de atribuição, primeiro o lado direito é processado. Se o resultado for um valor estático, o valor da variável é modificado e

```
case command of

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

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

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

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

Figura 2.4: Método *Online* para Especialização de Comandos FCL.

nenhum código é gerado. Se for dinâmico, a expressão especializada é utilizada na construção do comando de atribuição que irá aparecer no programa residual; além disso, a variável passa a ser classificada como dinâmica. A notação utilizada para modificação do valor de uma variável é `vs[`$X \mapsto$ `(type,val)]`, que significa: retorne uma nova lista `vs'` idêntica a `vs`, com exceção do valor da variável X, que é alterado para `(type,val)`.

2.4 Métodos *Offline*

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

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

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

Vamos mostrar a execução desses processos usando novamente a função Power da Figura 2.1.

2.4.1 Análise de Tempo de Definição

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

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

A computação da divisão das variáveis pode seguir dois métodos:

1. iteração até atingir um ponto fixo;

2. resolução de restrições (*constraint solving*).

O primeiro método é o mais simples e utilizado pela maioria dos avaliadores parciais mais antigos. Nesta seção utilizaremos apenas esse método, que executa diversos passos sobre o programa, até atingir umponto fixo na divisão das variáveis. Referências sobre análise de tempo de definição usando resolução de restrições podem ser encontradas em [7, 8, 10].

Se a função Power, do Figura 2.1, vai ser especializada com relação a n, o algoritmo de iteração até atingir um ponto fixo é disparado com a especificação $[n \mapsto S, x \mapsto D]$, indicando que n é estático (S) e x é dinâmico (D). A seguinte *divisão inicial* é computada:

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

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

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

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

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

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

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

Na Figura 2.5, é exibido o código anotado da função Power, a partir da divisão computada acima.. A codificação é feita usando-se uma linguagem de dois níveis [35, 34]. As estruturas dinâmicas aparecem sublinhadas.

As variáveis não precisam ser anotadas porque sua classificação pode ser obtida diretamente da divisão computada. Todos os comandos de atribuição

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

Figura 2.5: Programa Anotado.

cujo lado esquerdo é p ou x são anotados como residualizáveis, pois essas variáveis são dinâmicas. No primeiro bloco, a constante 1 é anotada, indicando que irá aparecer no código residual. Como visto na Seção 2.3, sempre que um valor estático aparece em um contexto dinâmico, ele sofre um *lift*. Assim, a anotação da constante 1 é equivalente a escrever `lift(1)`, mas neste caso a operação de *lift* está *compilada* no código anotado. Por enquanto, vamos deixar a compressão de transições de lado, assim todos os `goto` são anotados.

2.4.2 Geração dos Estados Alcançáveis

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

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

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

$$
\begin{aligned}
& (b0, [n \mapsto 2]) \\
\rightarrow \quad & (b1, [n \mapsto 2]) \\
\rightarrow \quad & (b2, [n \mapsto 2]) \\
\rightarrow \quad & (b3, [n \mapsto 2]) \\
\rightarrow \quad & (b1, [n \mapsto 1]) \\
\rightarrow \quad & (b2, [n \mapsto 1]) \\
\rightarrow \quad & (b4, [n \mapsto 1])
\end{aligned}
$$

$\rightarrow \quad (b1, [n \mapsto 0])$
$\rightarrow \quad (b5, [n \mapsto 0])$
$\rightarrow \quad (halt, [n \mapsto 0])$

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

2.4.3 Especialização dos Pontos de Programa

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

Por exemplo, no bloco

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

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

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

2.4.4 Compressão de Transições

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

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

2.4.5 Especialização Offline: Três Passos em Paralelo

Um algoritmo para avaliação parcial de programas usando um método *offline* tem o mesmo formato geral proposto para os métodos *online*, exibido na Figura 2.3. A diferença reside em como os comandos são tratados, refletindo a opção entre:

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

No método *online*, o algoritmo da Figura 2.3 tinha como entradas o programa original (`program`), o rótulo do bloco inicial do programa ($\texttt{pp}_0$) e o valor inicial das variáveis ($\texttt{vs}_0$). O método *offline* precisa dessas mesmas entradas, e de uma quarta, que é a divisão das variáveis computada pela BTA. Além disso, `program` é o programa anotado produzido pela BTA, e não mais o programa original.

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

Como fizemos na Seção 2.3, o código *offline* para processar cada comando FCL (`return`, atribuição, desvio condicional e incondicional) é apresentado separadamente do *loop* principal do algoritmo de especialização. Esse código pode ser observado na Figura 2.6.

As funções utilizadas na Figura 2.6 para avaliar expressões são `eval` e `reduce`. A função `eval` só é aplicada a expressões classificadas pela BTA como estáticas, retornando uma constante resultante da avaliação dessas expressões. A função `reduce`, por outro lado, só é aplicada a expressões classificadas como dinâmicas. O resultado da aplicação de `reduce` a uma expressão é uma nova expressão, onde todos as operações estáticas foram computadas. Isso configura uma política completamente diferente da adotada na abordagem *online*, onde a avaliação de uma expressão poderia resultar em um valor estático ou dinâmico, necessitando da propagação de valores com *tags*.

```
case command of

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

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

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

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

Figura 2.6: Método *Offline* para Especialização de Comandos FCL.

2.5 Comparação Entre Métodos *Online* e *Offline*

Para tecer comparações entre as duas abordagens propostas para avaliação parcial, vamos observar os programas residuais produzidos para a função Power, nas Seções 2.3 e 2.4.

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

```
p := 1;
```

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

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

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

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

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

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

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

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

Da mesma maneira que a verificação de tipos estática, avaliação parcial *offline* tem a vantagem de ser mais eficiente e permite uma verificação mais

fácil de propriedades do programa residual, antes que ele seja gerado. Uma vez que a divisão das variáveis é executada uma única vez, elimina-se o esforço adicional de se lidar com *tags* durante a especialização.

Os primeiros avaliadores parciais usavam sempre métodos *online*. As técnicas *offline* receberam um grande impulso quando se buscou a compilação e geração de compiladores usando avaliação parcial. Essas tarefas envolvem a auto-aplicação do avaliador parcial, processo que introduz muitos problemas para a abordagem *online*, mas que puderam ser resolvidos de maneira satisfatória com os métodos *offline*. Um dos motivos pelos quais os métodos *offline* facilitam a auto-aplicação é a divisão do processamento em duas fases. O código do avaliador parcial que é submetido a si próprio consiste em um programa anotado mais simples, pois executa apenas a segunda fase do método (especialização). No Capítulo 4 discutiremos a auto-aplicação com mais detalhe.

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

2.6 Tópicos mais Avançados de Métodos *Offline*

2.6.1 Assegurando a Terminação

No exemplo desenvolvido na Seção 2.4, utilizamos um algoritmo simples para produzir uma divisão das variáveis em estáticas e dinâmicas, na fase de BTA. Uma variável é classificada com dinâmica quando depende de um valor dinâmico em algum ponto do programa, o que convencionamos chamar de *princípio da congruência*. Como veremos a seguir, essa estratégia simples pode levar o avaliador parcial a um *loop* infinito.

Observe o trecho de programa FCL a seguir, adaptado de [22]:

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

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

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

$(b0, [x \mapsto 0])$: `if y` $\neq$ `0 goto` $(b1, [x \mapsto 0])$ `else` $(b2, [x \mapsto 0])$
$(b1, [x \mapsto 0])$: `y := y - 1`
`goto` $(b0, [x \mapsto 1])$
$(b2, [x \mapsto 0])$: `...`

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

$(b0, [x \mapsto 1])$: `if y` $\neq$ `0 goto` $(b1, [x \mapsto 1])$ `else` $(b2, [x \mapsto 1])$
$(b1, [x \mapsto 1])$: `y := y - 1`
`goto` $(b0, [x \mapsto 2])$
$(b2, [x \mapsto 1])$: `...`

O processo continuaria indefinidamente. O conjunto de estados alcançáveis seria infinito. O problema é que, embora x dependa apenas de constantes e de si mesmo, o conjunto de valores que pode receber é ilimitado, pois os valores são computados sob um controle dinâmico. Uma solução é fazer a BTA classificar todas as variáveis calculadas sob controle dinâmico como dinâmicas.

O processo de se classificar uma variável como dinâmica, mesmo quando o princípio da conguência permite que seja estática, é chamado de *generalização*. Esquemas para resolver esse problema podem ser encontrados em [20, 37, 19].

2.6.2 BTA com Divisão *Pointwise*

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

Observe o seguinte trecho de código FCL:

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

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

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

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

2.6.3 BTA com Divisão Polivariante

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

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

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

Uma divisão pointwise congruente teria obrigatoriamente que classificar x como dinâmica nos pontos que seguem $b1$ no fluxo de controle. Assim, teria `b2:`(D, D). Podemos verificar que, se $b2$ for atingido a partir de $b0$, x poderia ser tratado como estática. Uma divisão *polivariante* consegue lidar com essa abordagem, associando a cada rótulo um conjunto de divisões: `b0:`$\{(S, D)\}$, `b1:`$\{(S, D)\}$ e `b2:`$\{(S, D), (S, S)\}$.

Capítulo 3

Exemplos

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

3.1 Casamento de Padrões

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

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

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

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

```
b7:  return i - M
b8:  return -1
```

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

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

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

Observe que as variáveis i e j são atualizadas apenas com resultados de operações envolvendo constantes e o valor de si próprias. Na Seção 2.6, quando discutimos tópicos relacionados à terminação do processo de avaliação parcial, mostramos que uma divisão congruente pode levar o processo a um *loop* infinito. No exemplo desta seção, uma análise mais elaborada mostra que j tem crescimento limitado por N, um valor conhecido, enquanto que i tem crescimento limitado por N, um valor desconhecido. Assim, é necessária uma generalização da variável i, ou seja, classificá-la como dinâmica, embora o princípio da congruência permita que seja estática.

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

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

A forma de apresentação do programa anotado é a mesma adotada na Seção 2.4, com as operações dinâmicas sublinhadas. Os comandos `goto` são

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

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

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

Inicialmente, o bloco

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

é gerado, não envolvendo nenhuma computação de valores estáticos. Lembrando do algoritmo apresentado nas Figuras 2.3 e 2.6, o estado $(b1, [0])$ é acrescentado a `pending`, o conjunto dos estados ainda não gerados. Assim, o bloco

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

é gerado em seguida. O terceiro bloco será

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

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

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

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

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

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

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

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

$$\begin{array}{ll}(b5, [0]): & \texttt{i := i + 1} \\ & \texttt{goto } (b2, [1])\end{array}$$

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

$$\begin{array}{ll}(b6, [0]): & \texttt{i := i - (-1)} \\ & \texttt{goto } (b1, [0])\end{array}$$

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

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

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

O programa residual age como um autômato, mudando de estado a cada vez que um dos caracteres do padrão é encontrado no texto, em seqüência. Se

houver falha no casamento, retorna ao estado inicial, neste caso, representado por $(b3, [0])$. Uma otimização que não havia sido ainda comentada foi conduzida no código: os estados $(b8, [0])$, $(b8, [1])$ e $(b8, [2])$ foram fundidos em um único, rotulado como $(b8, [0])$, pois o código gerado para os três é idêntico.

3.2 Casamento de Padrões - Segunda Versão

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

Apresentamos em seguida uma nova versão para o programa de casamento de padrões, onde o texto A é lido apenas de maneira seqüencial. Ou seja, a variável i sofre apenas incrementos. Para conseguir isso, vamos utilizar a seguinte informação:

$$\text{se } \begin{cases} \text{o texto } A \text{ é da forma } A_0 A_1 \ldots A_{N-1}; \\ \text{o caractere corrente do texto é indicado por } i; \\ \text{o padrão } p \text{ é da forma } p_0 p_1 \ldots p_{M-1}; \\ \text{e o casamento falhou na posição } j \text{ de } p, \text{ com } 0 \leq j \leq M-1 \end{cases}$$
$$\text{então } A_{i-j} A_{i-j+1} \ldots A_{i-1} = p_0 p_1 \ldots p_{j-1}.$$

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

O novo código é apresentado a seguir.

```
(p, M, A, N)
(b00)

b00:  i := 0
      j := 0
      goto b01

b01:  if j ≥ M goto b13 else b02
b02:  if i ≥ N goto b14 else b03
b03:  if p[j] = A[i] goto b04 else b05

b04:  i := i + 1
      j := j + 1
      goto b01
```

```
b05:  if j = 0 goto b06 else b07
b06:  i := i + 1
      goto b01
b07:  k1 := 1
      k2 := j
      j := 0
      goto b08
b08:  if k1 ≥ k2 goto b12 else b09
b09:  if p[j] = p[k1] goto b10 else b11
b10:  k1 := k1 + 1
      j := j + 1
      goto b08
b11:  k1 = k1 - j + 1
      goto b08
b12:  k1 := 0
      k2 := 0
      goto b01
b13:  return i - M
b14:  return -1
```

Boa parte do código é idêntica à primeira versão do programa para casamento de padrões: os blocos `b00` a `b04`, `b13` e `b14`. Os blocos `b05` e `b06` tratam o caso especial que consiste na falha de casamento do primeiro caractere do padrão.

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

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

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

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

Um trecho do programa anotado é exibido abaixo, correspondendo aos comandos que não aparecem na primeira versão do programa para casamento

de padrões. Observe que nenhuma operação dos blocos `b07` a `b12` é marcada para ser residualizada.

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

Para efeito de comparação, vamos supor novamente que o programa anotado vai ser especializado para os valores de $p =$ "abc" e $M = 3$, como na seção anterior. O estado inicial será $(b00, [\text{"}abc\text{"}, 3, 0, 0, 0])$, onde os três últimos valores da lista de valores das variáveis correspondem às variáveis j, $k1$ e $k2$, respectivamente. Como p e M não sofrem alterações, vamos simplificar a representação dos estados desconsiderando essas variáveis.

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

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

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

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

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

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

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

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

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

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

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

O resultado acima leva a uma conclusão até certo ponto surpreendente. O programa original se baseou em um algoritmo ingênuo, *força-bruta*. Entretanto, o programa residual tem eficiência equivalente ao método KMP (Knuth-Morris-Pratt) de casamento de padrões, para um padrão específico. Esse resultado foi apresentado pela primeira vez em [11], onde os autores utilizaram um avaliador parcial para uma linguagem funcional.

Para finalizar este exemplo, vamos discutir a conveniência de se atribuir o valor 0 às variáveis $k1$ e $k2$ no bloco `b12`. Se isso não fosse feito, o estado $(b12, [0,2,2])$ geraria um desvio para $(b1, [0,2,2])$. O bloco correspondente a $(b1, [0,2,2])$ deveria ser então gerado, juntamente com uma longa seqüência de outros blocos. Entretanto, o código de $(b1, [0,2,2])$ é equivalente ao do bloco $(b1, [0,0,0])$. Isso acontece porque as variáveis $k1$ e $k2$ são definidas e utilizadas em um trecho do programa, mas estão "mortas" em outros trechos. Uma forma geral de se resolver esse problema é fazer uma análise de variáveis vivas e mortas, especializando cada bloco somente com relação às variáveis vivas no mesmo. A tarefa de identificar as variáveis vivas de um bloco envolve análise de fluxo de controle do programa [1, 33]. A atribuição do valor 0 às variáveis, no exemplo, serviu como um truque para evitar a geração desnecessária de estados, supondo que a análise de variáveis vivas não está disponível.

3.3 Interpretador para Máquina de Turing

Este exemplo exibe um interpretador de uma versão da *Máquina de Turing*. A máquina possui as seguintes instruções:

`right`, `left`, `write` a, `goto` i, `if` a `goto` i.

Um estado é caracterizado por:

- um valor indicando a próxima instrução I_i, que será executada no próximo passo;
- uma fita infinita cujas células podem armazenar elementos do conjunto $\{0, 1, B\}$, onde B é "branco";

- um valor que indica a posição da cabeça de leitura/gravação, ou seja, qual célula da fita está sendo analisada no momento.

Apenas um número finito de células da fita possui valor diferente de B e inicialmente a cabeça indica a primeira célula com essa característica, se houver.

A instrução `write` a altera o conteúdo da célula analisada para 0, 1 ou B, conforme o valor de a; `right` desloca a cabeça uma célula para a direita; `left` desloca a cabeça uma célula para a esquerda; `goto` i desvia o fluxo do programa para a instrução I_i; `if` a `goto` i é um desvio que só ocorre se o conteúdo da célula analisada for a. A máquina pára quando o fluxo é desviado para um rótulo inexistente.

O programa exemplo apresentado a seguir, extraído de [22], pode provocar uma alteração na fita ou entrar em *loop* infinito, se a fita não contiver nenhum símbolo 0. Vamos chamar esse programa de MT1:

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

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

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

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

```
[
   [0 "ifgoto" 0 3]
   [1 "right"]
   [2 "goto" 0]
   [3 "write" 1]
]
```

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

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

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

da lista L, `tl(`L`)` retorna o resto da lista L, descartando o primeiro elemento e `cons(`e`,`L`)` retorna uma nova lista onde e é o primeiro elemento.

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

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

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

O programa anotado associado a Δ é exibido abaixo:

```
(prog, dir)
(init)
```

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

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

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

Para exemplificar nossa notação simplificada, observe o ponto de programa jump no código do interpretador. As únicas variáveis vivas, além de `prog` (que não sofre alteração), são `progrest` e `prox`. Assim, um estado associado a esse ponto de programa será representado por $(jump, [progrest \mapsto \ldots, prox \mapsto \ldots])$.

Se `progrest` indicar o programa a partir da instrução 1, a representação será $(jump, [progrest \mapsto 1, prox \mapsto \ldots])$. Se `progrest` for a lista vazia, será representado por um rótulo não existente no programa MT.

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

$(init, [\ ])$: `esq := []`
`goto` $(loop,\ [progrest \mapsto 0])$

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

$(cont4,\ [progrest \mapsto 1, instr \mapsto [0\ \text{“}ifgoto\text{”}\ 0\ 3], op \mapsto \text{“}ifgoto\text{”}])$:
`goto` $(do\text{-}if,\ [progrest \mapsto 1, instr \mapsto [0\ \text{“}ifgoto\text{”}\ 0\ 3]])$

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

$(do\text{-}if,\ [progrest \mapsto 1, instr \mapsto [0\ \text{“}ifgoto\text{”}\ 0\ 3]])$:
`if 0 = hd(dir) goto`
$(jump,\ [progrest \mapsto 1, prox \mapsto 3])$
`else`
$(loop,\ [progrest \mapsto 1])$

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

$(jump,\ [progrest \mapsto 1, prox \mapsto 3])$:
`goto` $(loop,\ [progrest \mapsto 3])$

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

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

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

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

```
(loop, [progrest ↦ 1]):
    goto (do-right, [progrest ↦ 2, instr ↦ [1 "right"]])
(do-right, [progrest ↦ 2, instr ↦ [1 "right"]]):
    esq := cons (hd(dir), esq)
    dir := tl (dir)
    goto (loop, [progrest ↦ 2])
(loop, [progrest ↦ 2]):
    goto (do-goto, [progrest ↦ 3, instr ↦ [2 "goto" 0]])
(do-goto, [progrest ↦ 3, instr ↦ [2 "goto" 0]]):
    goto (jump, [progrest ↦ 3, prox ↦ 0]])
(jump, [progrest ↦ 3, prox ↦ 0]]):
    goto (loop, [progrest ↦ 0])
```

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

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

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

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

Vejamos como isso aconteceu:

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

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

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

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

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

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

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

No Capítulo 4 veremos uma generalização e extensões dessa conclusão.

Capítulo 4

Geração de Geradores de Programas

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

Seguindo a notação utilizada em [22], se P é um programa escrito na linguagem L, $[\![P]\!]_L$ é uma função que denota a sua semântica. Quando não for importante, poderemos omitir o subscrito L da notação. Sendo assim, a definição equacional do avaliador parcial `mix` é a seguinte:

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

O programa P, quando aplicado às entradas, in_1 e in_2, produz a saída *out*. O avaliador parcial `mix`, quando aplicado a P e parte de sua entrada (in_1), produz um novo programa identificado como P_{in_1}. O programa residual P_{in_1} produz a mesma saída *out*, quando a entrada restante (in_2) é submetida a ele. A avaliação parcial é vantajosa quando in_2 varia mais do que in_1 e P_{in_1} executa mais rápido sobre in_2 do que P, sobre in_1 e in_2.

As linguagens envolvidas são:

L **:** usada para implementar o avaliador parcial mix;

S **:** a linguagem fonte dos programas submetidos ao avaliador parcial; e

T **:** a linguagem objeto dos programas especializados produzidos.

Geralmente, S e T são idênticas, mas existem casos onde elas são distintas. Nos exemplos apresentados no Capítulo 2, as linguagens S e T são a linguagem de fluxogramas FCL. Não discutimos em qual linguagem o próprio avaliador

parcial estaria implementado, embora tenhamos apresentado um código para mix na Figura 2.3, usando uma linguagem algorítimica. Com pouco esforço, os algoritmos das Figuras 2.3, 2.4 e 2.6 podem ser traduzidos para a linguagem FCL.

4.1 Compilação e Geração de Compiladores

Como o avaliador `mix` é um programa com duas entradas, ele pode servir de entrada para si próprio. Futamura foi o primeiro a sugerir essa abordagem, no que passou a ser conhecido como *três projeções de Futamura* [15]. Nesta seção apresentaremos a primeira projeção.

Supondo *int* um interpretador de uma linguagem qualquer, escrito em S, a primeira projeção de Futamura mostra que compilação através de avaliação parcial sempre gera programas corretos:

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

Assim temos $target = [\![mix]\!]\,(int, source)$, ou seja, o programa objeto é resultado da avaliação parcial de um interpretador com relação a um programa fonte específico. Pudemos observar a aplicação desse procedimento no exemplo da Seção 3.3, quando realizamos compilação de um programa escrito na linguagem da Máquina de Turing para a linguagem FCL.

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

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

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

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

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

Observe que havíamos feito a suposição de que S era a linguagem fonte dos programas submetidos a `mix`. No caso da auto-aplicação, isso quer dizer que o próprio `mix` deve ser escrito na linguagem S. Nos exemplos do Capítulo 2, tanto a linguagem fonte dos programas submetidos a `mix`, quanto a linguagem

dos programas residuais produzidos, eram a linguagem FCL. Para podermos aplicar os procedimentos descritos nesta seção, seria necessário codificar `mix` em FCL.

O primeiro avaliador parcial auto-aplicável foi construído por Jones, Sestoft e Sondergaard, para uma linguagem de equações recursivas de primeira ordem. A primeira versão [23] requeria anotações prévias introduzidas pelo usuário, mas uma versão seguinte [24] era completamente automática.

Jorgensen realizou experimentos que envolviam a geração de um compilador para uma linguagem funcional de avaliação *lazy*, usando um avaliador parcial escrito em uma linguagem funcional de avaliação estrita [26]. Os experimentos mostraram que a velocidade de execução do código compilado foi equivalente ao produzido por compiladores comerciais.

A auto-aplicação de mix pode ir ainda mais longe. A terceira projeção de Futamura envolve geração de geradores de compiladores:

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

O programa `cogen` é chamado de gerador de compiladores, porque recebe um interpretador para uma linguagem L como entrada, produzindo um compilador de L para a linguagem dos programas residuais de `mix`.

Muitos experimentos envolvendo a geração automática de geradores de compiladores `cogen`, usando a auto-aplicação de um avaliador parcial, foram bem sucedidos. A maioria tinha como linguagem fonte uma linguagem não tipada [24, 16, 12, 25, 26, 18].

4.2 Geração de Extensões

Na realidade,o programa `cogen` apresentado na seção anterior é mais do que um gerador de compiladores. Se `cogen` for aplicado a um programa P qualquer, podendo ser um interpretador ou não, produz um *gerador de extensões* para P. Um gerador de extensões de um programa P é um programa P_{gen} que, quando executado com um valor in_1 para a primeira entrada de P, gera um programa residual P_{in_1}. O programa P_{in_1} é o resultado da avaliação parcial de P com valor in_1 para a primeira entrada.

Para facilitar o entendimento,vamos apresentar um exemplo onde um gerador de extensões simples é produzido. Para isso, vamos utilizar o primeiro exemplo deste capítulo, que é a função Power escrita em C, exibida na Figura 1.1. Power possui duas entradas, designadas n e x. Um gerador de extensões para Power é um programa que, quando recebe um valor in_1, produz uma função $Power_{in_1}$, resultado da especialização de Power com respeito a $n = in_1$. Um gerador de extensões Power_gen para a função Power é exibido a seguir.

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

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

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

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

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

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

Na década de 90, uma abordagem que tornou-se popular foi a de escrever um gerador de geradores de extensões `cogen` à mão [27, 4, 9], ao invés de se construir um avaliador parcial `mix`. O gerador `cogen` pode ser utilizado para realizar avaliação parcial de um programa P de modo tradicional. Para isso, basta gerar uma extensão de P, então aplicar essa extensão a um valor específico, produzindo um programa especializado. Por outro lado, vimos que `cogen` pode ser automaticamente gerado, a partir da auto-aplicação de `mix`. Assim, as duas abordagens parecem ser equivalentes. Então por que razão construir um gerador de geradores de extensões ao invés de um avaliador parcial auto-aplicável? Em [32], as seguintes razões são enumeradas:

1. O gerador de geradores de extensões pode ser escrito em outra linguagem, de nível mais alto, do que a linguagem dos programas que ele processa. Por outro lado, um avaliador parcial auto-aplicável deve ter o poder de processar o seu próprio texto.

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

3. Um avaliador parcial deve conter um interpretador, o que pode ser um problema sério para linguagens fortemente tipadas, como será discutido a seguir. O gerador de geradores de extensões, nem as extensões geradas, precisam conter um interpretador.

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

Na primeira projeção de Futamura, temos

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

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

Na segunda projeção de Futamura, temos

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

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

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

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

O gerador de geradores de compilador *cogen* tem uma execução ineficiente, e além disso, os compiladores gerados por ele também são ineficientes. O fato de conter um tipo universal para tratar todos os tipos da linguagem faz com que um programa de uma linguagem fortemente tipada se comporte como o de uma linguagem com tipagem dinâmica, perdendo assim as vantagens de eficiência das linguagens fortemente tipadas.

Um gerador de geradores de extensões escrito à mão transforma um programa escrito em uma linguagem L em outro da mesma linguagem. Assim não precisa conter um interpretador, e um compilador pode ser gerado sem auto-aplicação. Os geradores de extensões produzidos, bem como as próprias extensões, não herdam nenhum mecanismo para tratamento de um tipo universal.

As conclusões que se pode tirar são as seguintes:

- Para linguagens não tipadas, resultados satisfatórios podem ser conseguidos na geração de compiladores dirigida por semântica, usando auto-aplicação de um avaliador parcial.
- Para linguagens fortemente tipadas, é mais adequado construir `cogen` à mão. A avaliação parcial tradicional pode ser conduzida como descrevemos anteriormente, e é possível a geração de compiladores mais eficientes.

Capítulo 5

Leitura Adicional

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

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

O texto base para entendimento de avaliação parcial de programas é o livro de Jones, Gomard e Sestoft [22]. O livro cobre tópicos iniciais, como as definições básicas dos conceitos envolvidos, algoritmos para implementação de avaliadores parciais e exemplos, de forma bastante didática. Trata também aspectos relacionados à especialização de programas escritos em linguagens de paradigmas diversos: imperativo, funcional e lógico. Os capítulos finais apresentam tópicos mais avançados, como garantia de terminação da avaliação parcial, supercompilação etc.

Um segundo livro foi publicado sobre o assunto três anos depois, tendo como autores Danvy, Glück e Thiemann [13]. Na realidade, trata-se de uma reunião de artigos completos, procurando resumir o estado da arte e as perspectivas futuras da avaliação parcial de programas.

O artigo de Mogensen e Sestoft [32] é ideal para iniciantes em avaliação parcial, pois é um tutorial mais atualizado. Em poucas páginas, aborda a maioria dos aspectos mais importantes relacionados ao tema. Em particular, trata, com muito mais detalhe que o livro de Jones, Gomard e Sestoft, a geração à mão de geradores de geradores de extensões. Algumas referências bibliográficas citadas são mais recentes.

A Escola de Verão de 1998 do Departamento de Ciência da Computação da Universidade de Copenhagen (DIKU) abordou o tema: *Avaliação Parcial - Teoria e Prática*. Os textos dos seminários apresentados no evento são também uma leitura muito interessante. Infelizmente, os anais não foram publicados por completo, cabendo ao leitor interessado uma pesquisa no *site* do grupo

de pesquisa TOPPS: www.diku.dk/research-groups/TOPPS. O material não publicado inclui os textos e transparências utilizadas por John Hatcliff, que serviram como umas das principais bases do texto apresentado neste capítulo, em especial as seções 2.3 e 2.4.

O grupo TOPPS estuda aspectos relacionados à manipulação semântica de programas e compreende boa parte dos pesquisadores citados nas referências deste documento. No *site* do grupo pode-se obter cópias eletrônicas de muitos dos artigos citados e outras informações importantes.

Bibliografia

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

[2] L. Andersen. C program specialization. Master's thesis, DIKU, University of Copenhagen, Denmark, December 1991. Student Project 91-12-17.

[3] L. Andersen. C program specialization. Technical Report 92/14, DIKU, University of Copenhagen, Denmark, May 1992.

[4] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[5] P. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[6] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[7] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, 1993. DIKU Research Report 93/22.

[8] L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 61–71, 1994.

[9] L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September, 1994. (Lecture Notes in Computer Science, Vol. 844)*, pages 198–214. Berlin: Springer-Verlag, 1994.

[10] L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3):191–208, September 1995.

[11] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.

[12] C. Consel and S. Khoo. Semantics-directed generation of a prolog compiler. Technical Report YALEU/DCS/RR-781, Yale University, New Haven, Connecticut, May 1990.

[13] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

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

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

[16] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, Amsterdam: North-Holland, 1989.

[17] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Goedel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[18] R. Heldal. Generating more practical compilers by partial evaluation. In R. Heldal, C. Kehler Holst, and P. Wadler, editors, *Functional Programming, Glasgow 1991*, pages 158–163. Berlin: Springer-Verlag, 1992.

[19] C. Holst. Finiteness analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 473–495. ACM, Berlin: Springer-Verlag, 1991.

[20] N. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. Amsterdam: North-Holland, 1988.

[21] N. Jones, C. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *1990 International Conference on Computer Languages, New Orleans, Louisiana, March 1990*, pages 49–58. New York: IEEE Computer Society, 1990.

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

[23] N. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.

[24] N. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[25] J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992. Student Project 92-1-4.

[26] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992*, pages 258–268. New York: ACM, 1992.

[27] J. Launchbury and C. Holst. Handwriting cogen to avoid problems with static typing. In *Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218, 1991.

[28] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992. Available from `ftp.diku.dk` as file `pub/diku/semantics/papers/D-152.ps.Z`.

[29] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[30] T. Mogensen. Self-applicable online partial evaluation of pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*. New York: ACM, 1995.

[31] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *LOPSTR 92. Workshops in Computing*. Berlin: Springer-Verlag, Jan. 1993.

[32] T. Æ. Mogensen and P. Sestoft. Partial evaluation. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.

[33] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Franciso, CA, USA, 1997.

[34] F. Nielson and H. R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56:59–133, 1988.

[35] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science **vol. 34**. Cambridge University Press, 1992.

[36] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991. Report TRITA-TCS-9101.

[37] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. Amsterdam: North-Holland, 1988.

6 Avaliação Parcial de Programas

Flávia Peligrinelli Ribeiro

Implementação de Abstrações de Controle Paralelo em Linguagens Orientadas por Objeto

Fernando Caixeta Sanches[1], Flávia Peligrinelli Ribeiro[1], Wagner Meira Jr.[1], Mariza A. S. Bigonha[1]

[1] Departmento de Ciência da Computação
Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627
Belo Horizonte, MG, Brasil
{caixeta, flavia, meira, mariza}@dcc.ufmg.br

Resumo—
Um dos maiores desafios na paralelização de aplicações e a otimização do código paralelizado de forma a melhor explorar os recursos que o ambiente de execução provê. Esse desafio e uma constante durante o ciclo de vida uma aplicação paralelizada, seja por mudanças nos parâmetros de entrada ou nos recursos disponíveis. Uma estratégia para facilitar a paralelização de aplicações e utilizar mecanismos, tais como controle de abstração, que permitam expressar as oportunidades de paralelismo dessas aplicações, sem que se defina a sua implementação, o que as torna independentes de plataforma de execução.

Neste artigo discutimos e investigamos a utilização de mecanismos de abstração de controle implementados sobre linguagens orientadas a objeto. Apresentamos uma hierarquia de classes Java que implementa construções comuns como cobegin e forall e mostramos a sua utilização na paralelização de um algoritmo *branch-and-bound* para o problema do caixeiro viajante.

Keywords— **abstração de controle, orientação por objeto, linguagens paralelas, java, portabilidade**

I. Introdução

Aplicações normalmente apresentam mais oportunidades de paralelização que qualquer máquina existente possa efetivamente explorar. Embora uma dada aplicação possa apresentar implementações paralelas eficientes em várias arquiteturas, incluindo processadores vetoriais, multiprocessadores baseados em memória compartilhada e multicomputadores de memória distribuída, cada classe de aplicação pode explorar um subconjunto de oportunidades de paralelização diferente. Programadores normalmente exploram apenas as oportunidades de paralelismo passíveis de serem implementadas na máquina alvo, ignorando outras possíveis implementações. Embora essa estratégia possa resultar em implementações eficientes para arquiteturas específicas, pode ser difícil conseguir paralelizações eficientes para outros ambientes de execução, onde os recursos de paralelização ou demandas computacionais dependentes dos dados sejam diferentes. Em suma, a maioria dos programas paralelos são uma descrição do paralelismo que é o mais apropriado para as premissas originais a respeito da arquitetura sendo utilizada. Durante a implementação de programas paralelos, programadores sempre procuram a estratégia de paralelização que permita obter a melhor relação custo-benefício para a implementação. Os possíveis benefícios incluem menor tempo de execução em virtude do paralelismo efetivamente explorado e maior facilidade para balancear a carga em virtude da menor granularidade das tarefas. Por outro lado, exemplos de custos incluem a computação adicional introduzida pelo gerenciamento, comunicação e sincronização entre processos. Quaisquer variações nesses custos afetam significativamente as decisões de implementação quando da paralelização dos programas. Como há diversas situações inerentes ao ciclo de vida de uma aplicação é necessária uma reavaliação das decisões de implementação:

- quando paralelizando uma aplicação pela primeira vez;
- quando ocorrem mudanças no *hardware* utilizado, tanto quantitativa, como por exemplo um novo processador, quanto qualitativa, como por exemplo, aumento no número de processadores ou memória disponíveis;
- quando a aplicação é portada para uma nova arquitetura;
- quando se deseja explorar *hardware* específico, como processadores vetorias; e
- quando é necessário otimizar um programa para uma dada classe de valores de entradas. Por exemplo, programas que operam sobre grafos densos e esparsos.

Assim, a facilidade para realizar essas tarefas é diretamente proporcional à facilidade de selecionar e implementar estratégias de paralelização alternativas. Abstração de controle [IcB 94] é um processo pelo qual programadores podem definir novas estruturas de controle, especificando restrições sobre a ordem de execução das operações separadamente da sua implementação. O uso de abstrações de controle não apenas provê uma maior

liberdade para o programador, como também permite representar as oportunidades de paralelização de um algoritmo. Uma vez que abstrações de controle separam a especificação da implementação, uma dada construção pode ter várias possíveis implementações, cada uma explorando um subconjunto das oportunidades de paralelismo admitidas pela construção. Via seleção de implementações usando diretivas, um programador pode variar as decisões de implementação sem alterar o código fonte. Esta abordagem produz programas que explicitam muitas das oportunidades de paralelismo do algoritmo, cujo desempenho pode ser melhorado simplesmente pela seleção de estratégias entre as várias implementações possíveis para as construções de controle utilizadas. Abstrações de controle já foram implementadas utilizando facilidades de linguagens funcionais e linguagens imperativas como por exemplo LISP [Crl 91, Mc 66] e CLU [LBS 81], mas ainda não foram implementadas em linguagens orientadas por objetos, onde propriedades como herança e polimorfismo facilitariam ainda mais a implementação destas construções e a determinação da estratégia de paralelização mais adequada para um algoritmo e seu ambiente de execução. Neste artigo discutimos a implementação de mecanismos de abstração de controle em linguagens orientadas por objeto. Na Seção II definimos formalmente o conceito de abstração de controle e os seus requisitos básicos. A Seção III discute a implementação desse conceito em Java [GJS 96]. A paralelização de um algoritmo de *Branch-and-Bound* utilizando abstração é mostrada e discutida na Seção IV. A última seção apresenta as conclusões e trabalhos futuros.

II. Abstração de Controle

Adaptabilidade arquitetural é a facilidade com que programadores podem implementar, otimizar ou portar um programa para um dado ambiente de execução. Grande parte dos programas seqüenciais se adaptam facilmente a novos ambientes de execução porque o código fonte se baseia em poucas premissas a respeito do ambiente de execução. Programas paralelos, por outro lado, têm a sua implementação significativamente afetada pelos custos adicionais que advém do processo de paralelização. Quando um ambiente de execução viola essas premissas, faz-se necessário reestruturar o programa de forma a evitar severa degradação de desempenho ou possibilitar a exploração de outras potenciais fontes de paralelismo. Como mencionado anteriormente, essa reestruturação pode ser complexa, em virtude de características implícitas do ambiente de execução. O nosso objetivo, portanto, é prover mecanismos que maximizem a adaptabilidade arquitetural de programas paralelos, mais especificamente, programas paralelizados explicitamente utilizando primitivas como fork, cobegin e forall [leB 94]. Uma vez que a expressão de paralelismo nessas linguagens é uma questão de controle de fluxo, devemos focar em mecanismos que permitam ampla exploração do paralelismo das aplicações. Entretanto, dada a variedade de aplicações passíveis de paralelização e a constante evolução das arquiteturas paralelas, parece ambiciosa a tarefa de tentar definir uma linguagem que permita adaptabilidade arquitetural com base em um conjunto pequeno e fixo de construções de controle de fluxo. Mais ainda, se o objetivo é incentivar programadores a explicitarem tantas oportunidades de paralelização quantas forem possíveis, nenhum conjunto limitado de construções será suficiente. Assim, é necessário criar um mecanismo que expresse precisamente o paralelismo de um algoritmo. Abstrações de controle provêm a flexibilidade e extensibilidade necessárias e permitem aos programadores criarem novas construções além das inerentes à linguagem de programação utilizada. Cada construção aceita, como parâmetro, um segmento de código para executar e seu ambiente de execução. A implementação de uma construção executa o código em uma ordem consistente com a sua definição. Para que uma linguagem de programação suporte a implementação de mecanismos de abstração de controle, seis mecanismos primitivos são indispensáveis:

Execução sequencial: deve ser possível definir uma ordem total de execução entre operações. Em linguagens imperativas, execução sequencial é o mecanismo padrão para ordenar operações que compõem um bloco básico, as quais são executadas na ordem que aparecem no programa.

Execução condicional: a linguagem deve permitir que predicados lógicos determinem a execução ou não de segmentos de código. Esse mecanismo também está presente em linguagens imperativas, como os `if` das linguagens C e Pascal.

Encapsulação de operações: a linguagem deve prover suporte para encapsular conjuntos de operações e restrições de ordem da sua execução. Mais uma vez, funções e procedimentos, inerentes às linguagens imperativas são um exemplo de como esse mecanismo pode ser implementado.

Chamada blocante de funções: uma função que chama outra função só continua a execução após a função chamada ser concluída. Mais uma vez, esse é o mecanismo padrão de linguagens imperativas.

Suporte à concorrência: uma função que chama outra função deve poder, se assim especificado, continuar a sua execução concorrentemente à função chamada. Em sistemas operacionais Unix, uma for-

ma de implementar tal mecanismo é por meio da primitiva `fork`. Uma outra forma de implementar tais mecanismos é via processos leves (*threads*), cuja execução concorrente é baseada em funções executadas em paralelo.

Suporte à sincronização: um mecanismo complementar ao suporte à concorrência é a possibilidade de sincronização concorrentes sincronizarem entre si, de forma a não ocorrerem condições de corrida e outros problemas que podem levar os programas paralelos a resultados inconsistentes. Semáforos, um mecanismo disponível em muitas das bibliotecas que implementam processos leves, é um exemplo.

Note que os mecanismos necessários para a implementação de abstrações de controle estão presentes em linguagens tradicionais como Ada, Modula-2, Java. Entretanto, somente alguns desses mecanismos estão presentes em linguagens como C, C++ e Pascal , além da não padronização de bibliotecas de *threads*[1]. Na próxima seção discutimos a implementação de mecanismos de controle de abstração em linguagens orientadas por objeto, mais especificamente Java, que além de possuir todos os mecanismos necessários para a implementação de abstrações de controle, tem como uma das suas principais características a portabilidade.

III. Usando Java para Implementar Abstração de Controle

Nesta seção descrevemos como mecanismos de abstração de controle podem ser implementados em Java e quais as vantagens adicionais que o uso de linguagens orientadas por objeto oferecem nesse sentido.

A. Java como Ambiente de Programação

Modularidade e abstração são considerados os dois mais importantes princípios na construção de grandes sistemas. O uso de modularização e abstração apropriados, nos auxilia a lidar com as complexidades inerentes de grandes problemas. Contudo, mesmo com a modularização apropriada, o projeto de todos os módulos a partir do zero é uma tarefa tediosa e consome muito tempo. A programação Orientada por Objeto aparece como uma metodologia atraente nesse sentido, porque ela propicia reuso de abstrações e módulos já definidos, ao invés de criar novas abstrações e implementar novos módulos para cada programa novo [Watt 90].

Considerando então que em um estilo de programação orientado por objetos *puro*, a unidade de modularização é uma implementação de um tipo abstrato de dados e que classe pode ser usada para definir os tipos abstratos de dados [CW 85]. Outro ponto muito importante nesse estilo de programação está relacionado com a forma em que conjuntos de conceitos relacionados são colocados juntos permitindo a produção de bibliotecas que são fáceis de entender devido ao relacionamento existente entre seus componentes e fáceis de serem estendidos pelo uso da herança. A herança provê um mecanismo para o desenvolvimento de uma família de abstrações organizadas como uma hierarquia de classes. Portanto, a organização hierárquica é considerada outro método eficiente para controlar a complexidade de sistemas.

Dentre as várias linguagens de programação projetadas especificamente para suportar orientação por objetos escolhemos Java para implementar as abstrações de controle paralelas por várias razões. Primeiro porque Java possui facilidades para programação concorrente e portanto serve como um bom estudo de caso. Segundo porque em Java, uma nova definição de classe possui no máximo uma classe pai, ou seja, ela possui um modelo único de herança. Além disto, Java é portável. Programas em Java são traduzidos para um código intermediário, *bytecodes*, que define a máquina virtual Java permitindo que qualquer máquina possuindo um interpretador de *bytecode* possa executar programas em Java. Esse fato faz com que o conceito de portabilidade seja estendido para a noção de mobilidade. Um interpretador de *bytecode* executando em um computador dentro de uma rede pode carregar e executar um programa ou uma classe de outro computador presente na rede [GC 98].

Em suma, a linguagem Java suporta todos os mecanismos primitivos necessários à abstração de controle, além de ser uma linguagem de alta portabilidade. O outra característica interessante é que o seu formato intermediário também é padronizado e a arquitetura das máquinas virtuais Java aberta, havendo, inclusive, várias implementações de domínio público, o que facilita obter informações que possam aumentar a eficiência das construções criadas e criar ferramentas que auxiliem a programação, tais como ferramentas para análise de desempenho [WM 97].

Como um estudo de caso para nosso problema, contudo, a facilidade mais importante em Java é a concorrência. Java suporta concorrência no nível de linguagem e não por meio de suas bibliotecas em tempo de execução, o que significa que a facilidade para concorrência é integrada com os objetos. Concorrência é obtida pela classe pré-definida *Thread*. Uma nova classe que herda de *Thread* pode executar concorrentemente com outras *treads*. As bibliotecas de "tempo-de-execução" provêem dois mecanismos que podem ser usados pelos programadores para criar *treads* paralelos para execução: uma

[1] Apenas recentemente foi determinado um padrão POSIX para *threads* que ainda não é satisfeito em muitas implementações existentes.

interface *Runnable* e uma classe *Thread*. A classe *thread* possui vários métodos que controlam a execução de um objeto *thread*, por exemplo, uma chamada ao método `start( )` de um *thread* faz com que o método `run( )` seja executado associado ao objeto. Existem duas formas de se criar um objeto *thread*. Para criar um objeto que pode rodar em um independente *thread* de execução, o objeto *Thread* pode ser estendido e fornecer o método `run()`. Outra forma de criar um objeto *thread* é estendendo a interface *Runnable*. *Runnable* define um único método abstrato chamado `run()`. Qualquer classe que implementa *Runnable* deve prover seu próprio método `run()`. Objetos desta classe são capazes de executar em um *thread* concorrentemente com outros *threads* [GC 98].

Uma razão para usar *Runnable* ao invés de *Thread* na definição de objetos *threads* está relacionado com o fato de que se tivermos uma classe que queremos executar, podemos definir uma nova classe que herda da classe existente e implementa *Runnable* ao mesmo tempo. Este é um exemplo de como Java oferece o uso de herança multipla.

B. *Implementação de uma Hierarquia de Abstrações de Controle*

Nesta seção discutimos como uma linguagem orientada por objeto, mais especificamente Java, pode ser usada para implementar mecanismos de abstração de controle. Assim, definimos uma hierarquia simples, onde o primeiro nível é representado pela classe *thread* (inerente a Java) e no segundo nível temos duas classes derivadas, cobegin e forall, que se diferenciam por instanciar em paralelo segmentos de código heterogêneos ou não, respectivamente. As subseções seguintes descrevem as implementações dessas classes derivadas.

B.1 Implementação CoBegin

Essa construção permite que dois blocos distintos executem concorrentemente, retornando o controle somente após o término de ambas as execuções.

A regra para execução correta, diz que ambas as execuções começam após o início do **cobegin** e retornam antes dele retornar e não há dependências entre os dois blocos.

Para garantir a validade das regras acima, o controle da execução (que pertence à *thread* principal), é passado para as duas *threads* que irão executar os blocos. Assim, ela fica esperando o término da execução das filhas para, então, reassumir o controle da execução.

Para implementar essa abstração, foi utilizado o conceito de interface (comando_cobegin) em java, uma vez que o corpo dos blocos deveria ser entregue ao método Cobegin. Da mesma forma, para garantir a correção das tranferencias de controle, foi utilizada o método *join*, que garante que a *thread* mãe espere o término da execução de sua(s) filha(s).

Abaixo, pode-se ver a implementação em alto nível do Cobegin:

```
cobegin(comando_cobegin stmt1, comando_cobegin
        stmt2)
{
    processo1 = new Thread(stmt1);
    processo2 = new Thread(stmt2);
    // Obriga a thread mae esperar que as duas
       filhas terminem suas execucoes
    processo1.join();
    processo2.join();
}
```

B.2 Implementação Forall

A construção permite que a execução do comando for seja realizada em paralelo, ou seja, as iterações são associadas a *threads* que irão realizar a devida computação.

Forall deve iniciar antes de qualquer iteração, e todas elas devem retornar antes dele, para que se garanta seu funcionamento correto. Da mesma forma, é preciso que as iterações sejam independentes umas das outras.

Pelo mesmo motivo do cobegin, aqui também foi utilizado o conceito de interface (comando_for) bem como a *thread* mãe (que cuida da execução do programa), deve esperar o término da execução das filhas (que cuidam da execução do corpo do for).

Abaixo, pode-se ver a implementação em alto nível do Forall:

```
forall(int inicio, int fim, comando_for body,
       int num_threads)
{
    Testes de consistência; // relacao num_threads
                               com num_iteracoes

    for(i=0; i|n_threads; i++)
        processos[i] = new thread(minha_parte,
                                  comeco, stmt);

    for(i=0; i|n_threads; i++)
        processos[i].join();
}
```

A utilização do paradigma de orientação por objetos foi adotada para a implementação das abstrações, uma vez que facilita a utilização dos componentes criados, bem como aumenta o reuso do código utilizado.

nodos a serem expandidos: se o limite inferior de um nodo for maior que que a melhor solução encontrada durante a execução não faz sentido que esse nodo seja expandido.

B. Aplicação das Abstrações

As abstrações de controle implementadas, foram introduzidas em pontos críticos da aplicação, de forma que se pudesse avaliar funcionalidade dessas no programa. Assim, **Cobegin** (paralelismo por tarefa) cuida da parte de expansão dos nodos, enquanto váios **forall** foram introduzidos nos caminhamentos na matriz de custos.

Assim, o algoritmo modificado pode ser visto como:

```
enquanto (ha nos com limite menor que global) {
    aresta = proxima aresta discriminante
    cobegin(inclui(aresta), exclui(aresta));
    se solucao atualiza limite global
}

inclui(aresta) {
   atualiza estado incluindo a aresta
   forall(nodos no grafo)
     se o nodo ja possui duas arestas incluidas
        forall(nodos no grafo)
          exclui as demais arestas
   cria estado e insere no conjunto
}

exclui(aresta) {
   atualiza estado retirando a aresta
   forall (nodos no grafo)
     se nao houver duas arestas nao excluidas
       return
   cria estado e insere no conjunto
}
```

Esse algoritmo foi escrito em java de forma a se beneficiar tanto da orientação por objetos quanto da portabilidade que ela provê.

Através dessas implementações, pretende-se facilitar ainda mais a programação em paralelo, uma vez que essa tem se tornado cada vez mais comum e necessária.

Assim, com a utilização das estruturas propostas na seção anterior, a paralelização do problema do Caixeiro Viajante se tornou transparente, uma vez que não foi preciso manipular as *threads* (que são tratadas dentro das estruturas), bastando apenas encontrar as partes paralelizáveis.

C. Discussão

Com a utilização da abstração de controle implementada, a utilização de paralelismo na solução do problema do Caixeiro Viajante se tornou transparente para

7 Implementação de Abstrações de Controle...

Estratégias para Otimização de Código Móvel

Marco Túlio de Oliveira Valente

8 Estratégias para Otimização de Código Móvel

Coleta de Lixo

Mark Alan Junho Song

1 Introdução

Coleta de Lixo (*Garbage Collection*) é um processo que permite o gerenciamento automático de memória de tal forma que entidades alocadas dinamicamente são liberadas, quando não mais necessárias, sem a intervenção do programador.

Todo o processo, neste trabalho, é baseado na alocação dinâmica de registros que podem possuir componentes que referenciam áreas de memória no *heap*. Como nem sempre é possível determinar, sem uma análise mais detalhada, se um registro está dinamicamente vivo procura-se usar uma abordagem mais conservativa: um registro está vivo se puder ser alcançado a partir de alguma variável de programa.

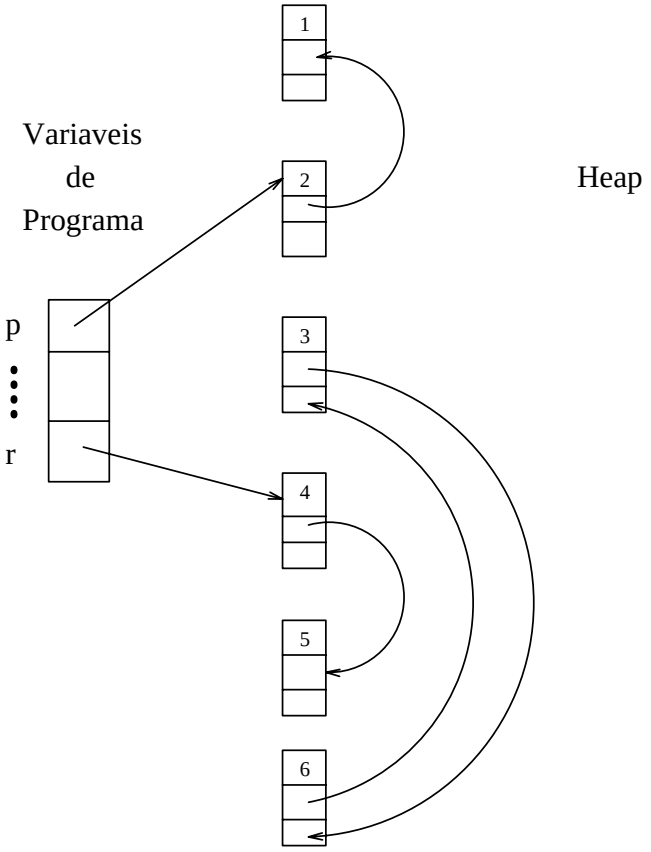


Figura 1: Alocação Dinâmica.

Considere o seguinte fragmento de programa:

```
struct no {
       int        key;
       struct no *left, *right;
};
```

```
main()
{
    no *p = (struct no *) malloc( sizeof(struct no) );
    no *r = p->right;

          . . . .
          . . . .
}
```

A Figura 1 mostra uma possível situação do heap após alocações. Note que os registros 1, 2, 4, 5 estão vivos uma vez que 1 e 2 podem ser são alcançados a partir de p enquanto 4 e 5 a partir de r. Os registros 3 e 6 estão mortos já que não podem ser acessados por nenhuma variável de programa.

O objetivo principal deste trabalho é apresentar técnicas e algoritmos que permitam uma coleta de lixo eficiente visto que tal processo, embora exija interação com o compilador na geração de código, não é realizado pelo mesmo mas sim por rotinas de suporte de execução ligadas ao código compilado.

2 Critérios de Projeto

Quando se trabalha com coleta de lixo alguns aspectos devem ser levados em consideração no projeto a ser desenvolvido. Os principais critérios a serem observados são:

- Qual o *overhead* envolvido? Como qualquer sistema, a coleta de lixo obviamente consome recurso seja na forma de memória adicional ou ciclos de CPU para realizar sua tarefa.
- O sistema permite compactação? Se o gerenciador de memória repetidamente alocar e liberar células de tamanho variado o *heap* tende a se fragmentar. Isto significa que uma célula pode não ser alocada por não existir um bloco contíguo livre de tamanho adequado. Tal fenômeno conhecido como fragmentação [Knuth, 1976] será abordado posteriormente.
- Qual a dimensão adequada do *heap* e quando efetuar uma coleta de forma a minimizar o custo?
- O sistema recupera estruturas cíclicas?
- O sistema opera em máquinas paralelas, ou em tempo real? Algumas técnicas exigem uma parada do processo para que a coleta possa ser efetuada, o que é inadmissível para aplicações de tempo real. Técnicas especiais são necessárias para que o coletor possa trabalhar em paralelo com o processo corrente. Não abordaremos neste trabalho tais técnicas. O leitor interessado poderá encontrá-las em [Dijkstra 1978].

A Seção 3 apresentará as principais técnicas de coleta de lixo: *Reference Counts Collection*, *Mark and Sweep Collection*, *Copying Collection*.

3 Técnicas de Coleta de Lixo

Para as técnicas apresentadas considere:

1. X é um registro e f_i um de seus campos que denota um apontador para uma área de memória alocada dinamicamente.
2. Toda área alocada dinamicamente corresponde a um registro de tal forma que $x.f_i$ é também um registro alocado no *heap*.
3. Um registro está vivo se puder ser alcançado a partir de alguma variável de programa.

3.1 Coleção de Contadores de Referência[1]

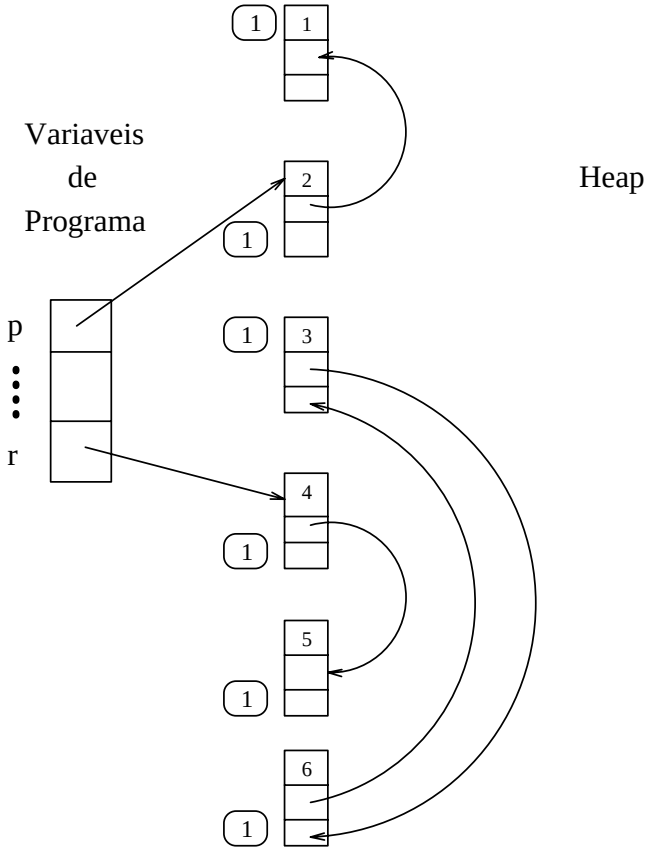


Figura 2: Reference Counts Collection.

Neste processo a coleta de lixo é feita diretamente mantendo um contador de referência armazenado no próprio registro. O compilador emite instruções extras sempre que uma área

[1] do inglês Reference Counts Collection

dinâmica p for armazenado em $x.f_i$. O contador de referência de p é incrementado, e o de $x.f_i$ previamente apontado decrementado. Se o contador de um registro r atingir zero, então r é colocado em uma lista de blocos livres e todos os registros para os quais r aponta são decrementados.

A técnica Coleção de Contadores de Referência é simples e fácil de ser implementada, mas apresenta dois grandes problemas:

1. Estruturas cíclicas podem não ser devolvidas para a memória. Note pela Figura 2 que existe um ciclo das células 3 e 6 que não são alcançáveis por nenhuma variável de programa. Mesmo assim, não são devolvidas à gerência de memória por possuir contadores de referência diferentes de zero.

2. O custo elevado no acesso aos contadores. Considerando a seguinte atribuição $x.f_i = p$, o programa executa em nível de instruções de máquina, o seguinte código:

```
z = x.fi
c = z.count
c = c - 1
z.count = c
if c = 0 goto PutOnFreeList ( z )
x.fi = p
c = p.count
c = c + 1
p.count = c
```

Segue abaixo uma implementação, simplificada, retirada do Compilador Visual C++ 5.0 esboçando um coletor de lixo baseado em contadores de referência.

```
Class RefCount {
Private:
    Int crefs;
Public:
    RefCount( )  { crefs = 0; }
    Void upcount( ) { ++crefs; }
    Void downcount( ) { if  !(--crefs) delete this; }
};

template <class T> class  Ptr {
Private:
    T* p;
Public:
    Ptr(T* p_) : P(p_)  { p->upcount ( ); }
    Ptr = (Ptr <T> p_) {
           This->p->downcount( );
```

```
            This->p = p_->p;
            P_->upcount( );
    }
    ~Ptr(void) { p->downcount( ); }
    operator T* (void) { return p; }
    T& operator *(void) { return *P; }
    T* operator ->(void) { return P; }
    Ptr& operator =(T* p_) {
                    p->downcount( );
                    p = p_;
                    p->upcount( );
                    return *this
    }
};
```

As classes RefCount e $Ptr < T >$ em conjunto fornecem um mecanismo simples de coleta de lixo para qualquer classe, derivada de RefCount, que pode suportar o *overhead* de adicionar um inteiro por instância.

Uma possível solução para o problema de ciclos é combinar a técnica de contador de referência com ocasionalmente a técnica *mark and sweep* que será apresentada na Seção 3.2. Como, em geral, as desvantagens deste método superam sua simplicidade dificilmente tal técnica é usada em gerenciamento automático de memória.

3.2 Mark and Sweep Collection

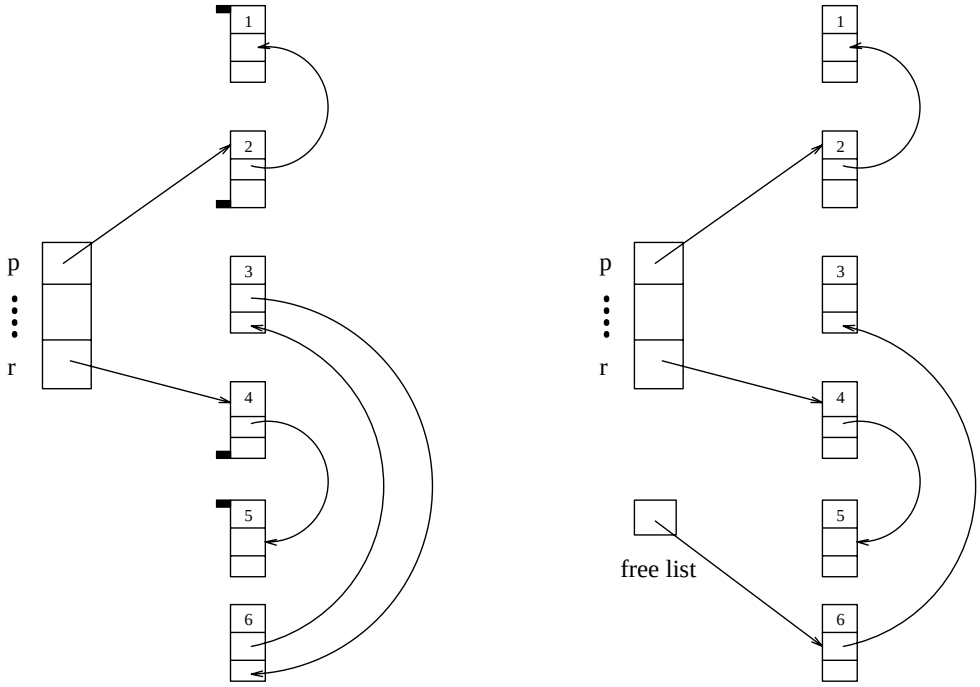
Variáveis de programa e registros alocados no *heap* formam um grafo direcionado onde as variáveis são raízes do grafo, os registros nós e os apontadores arestas conectando os respectivos nós. Um nó n é alcançável se existir um caminho começando por alguma raiz r que chegue em n.

Diferente da técnica apresentada anteriormente, *mark and sweep colletction* divide a coleta de lixo em duas etapas:

1. Efetua uma busca no grafo marcando todos os nós alcançáveis (vide Figura 3a). Todos os nós não marcados nesta etapa são considerados lixos devendo ser devolvidos a gerência de memória.

2. Percorre o *heap*, do primeiro endereço ao último, procurando por nós não marcados. Estes são armazenados em uma lista de blocos livres. Note que esta fase irá também desmarcar os nós, preparando-os para uma posterior coleta de lixo (vide Figura 3b).

Em geral esta técnica exige uma parada no processo corrente. Após a coleta, o programa continua sua execução normalmente. Todo registro alocado dinamicamente é obtido da lista de blocos livres. Uma boa hora para se efetuar uma coleta de lixo é quando a lista começar a ficar vazia. O algoritmo é apresentado a seguir:



(a) Mark (b) Sweep

Figura 3:

3.3 Algoritmo para Mark and Sweep

```
Mark phase:

     Function  Depth_First_Search ( x )
       If x is a pointer into the heap
          If record x is not marked
             Mark x
             for each field fi of record x
                 Depth_First_Search ( x.fi )
```

```
Sweep phase

     P = first address in heap
     While  p < last address in heap
     begin
            If record p is marked
               Unmark p
            Else let f1 be the first field in p
            begin
                p.f1 = freelist
                freelist = p
            end
            p = p + (size of record p)
     end
```

Custo da Coleta

É fácil verificar que a fase inicial executa em tempo proporcional ao número de nós marcados, ou seja, proporcional ao número de nós alcançáveis. A segunda fase executa em tempo proporcional ao tamanho do *heap*. Suponha R o número de nós alcançáveis expresso em palavras e H o tamanho do *heap*. O custo de uma coleta é então $c_1R + c_2H$. Como a cada coleta recupera-se $H - R$ palavras úteis, o custo amortizado é dado por: $(c_1R + c_2H)$ / $(H - R)$ instruções por palavras alocadas.

Se R for próximo de H o custo se torna proibitivo. Por outro lado se H for muito maior que R, então o custo é aproximadamente c_2. Tecnicamente uma boa medida é considerar R/H próximo de 0.5. Se após uma coleta esta razão for muito maior que 0.5, o coletor pode incrementar H solicitando mais memória ao sistema operacional.

Desvantagens

Note que o algoritmo utilizado na fase inicial é um caminhamento em profundidade recursivo. Tal algoritmo exige uma pilha que suporte o maior caminho de um nó alcançável no grafo. Como na pilha empilhamos registros de ativação, existindo um caminho de tamanho H, no pior caso, a pilha provavelmente será maior que todo o *heap*. Uma solução para este problema é manter a pilha explicitamente reduzindo-a para H palavras ao invés de H registros de ativação.

Um outro problema encontrado se relaciona a lista de blocos livres. Para uma lista simples é necessário armazenar em cada bloco seu tamanho para posterior utilização. Além disso, ao ser requisitado uma nova área de memória o alocador pode ser obrigado a percorrer toda a lista a procura de um bloco de tamanho adequado. Uma boa solução para o problema é a utilização de um arranjo de listas de tamanhos variados.

Se um programa solicita uma área de tamanho i basta utilizar o *head* dado por ListaBloco[i]. Se ListaBloco[i] estiver vazia basta alocar de ListaBloco[j], $j > i$. O Bloco restante $j - i$ é devolvido para a entrada ListaBloco[j-i].

Fragmentação

Mesmo considerando diversas listas de blocos livres tem-se ainda um outro problema: Como alocar uma nova área de tamanho i se apenas blocos de tamanho j, $j < i$, existirem? Este problema é denominado fragmentação e exige a compactação de blocos menores para a obtenção de uma área contígua de memória de maior tamanho.

3.4 Copying Collection

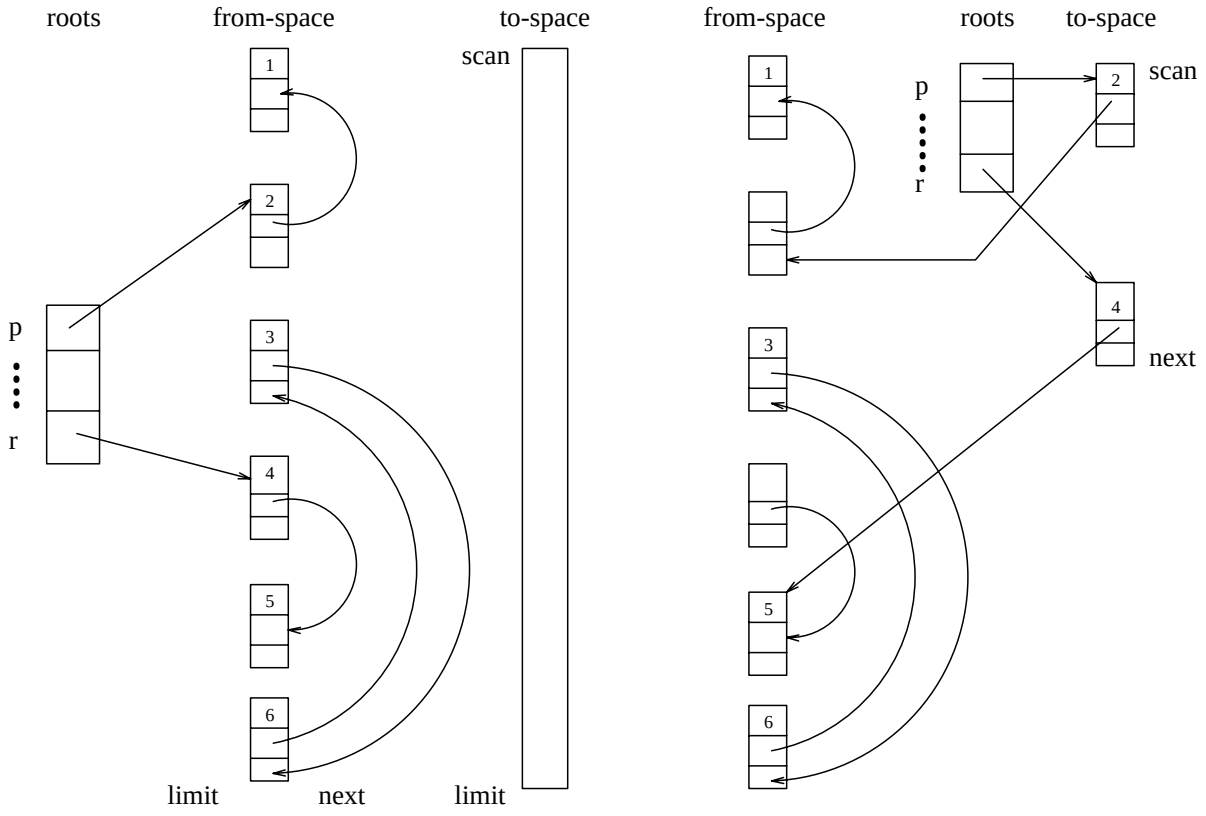
Copying Collection evita o problema de fragmentação percorrendo o grafo em uma área denominada *from-space* construindo uma cópia do mesmo em uma nova área do *heap* denominada *to-space*. A nova cópia gerada é compacta ocupando posições contíguas de memória sem fragmentação. A Figura 4 ilustra parte do processo antes e depois da cópia.

Note que o *heap* é divido em dois sendo controlado por elos apontadores *next* e *limit*. Antes da coleta, a área entre *next* e *limit* está disponível para a alocação de novos registros. Quando *next* atingir *limit* uma nova coleta é necessária. Neste caso, copia-se os registros vivos em *from-space* para *to-space* e seus papéis se invertem.

3.5 Algoritmo

Para inicializar uma coleta, o apontador *next* recebe o endereço da primeira posição de *to-space*. Um registro R é então copiado para a posição *next*, e *next* é incrementado pelo tamanho de R. O processo se repete até que todos os registros vivos sejam copiados de *from-space* para *to-space*. Segue abaixo o algoritmo completo:

```
function Forward ( p )
begin
    if p points to from-space
    then if p.f1 points to to-space
         then return p.f1
         else begin
              for each field fi of p
                  next.fi = p.fi
              p.fi = next
              next = next + (size of record p)
              return p.f1;
         end
    else return p
end
```



(a) Antes da Coleta

(b) Roots Fowarded

Figura 4:

```
scan = next = beginning of to-space
for each root r
    r = Forward ( r )
while scan < next
      for each field fi of record at scan
           scan.fi = Forward ( scan.fi )
      scan = scan + (size of record at scan)
```

Custo da Coleta

Verifica-se que o algoritmo é executado em tempo proporcional ao número de nós marcados, ou seja, proporcional ao número de nós alcançáveis. Como não existe uma fase posterior o custo da coleta é $c_1 R$. Levando em consideração que o *heap* é dividido em dois semi espaços, cada coleta obtém $(H/2 - R)$ palavras que podem ser alocadas antes de uma nova coleta. O custo amortizado é então $c_1 R$ / $(H/2 - R)$ instruções por palavras alocadas. A medida em que H cresce o custo da coleta tende a zero.

4 Coleta de Lixo nos Diferentes Paradigmas de Programação

Basicamente, as técnicas apresentadas anteriormente podem ser aplicadas em diferentes linguagens de programação. Esta seção introduz alguns pontos relevantes sobre a coleta de lixo em linguagens imperativa e orientada por objeto como C e C++.

Coleta de Lixo em C e C++

Um sistema de gerenciamento automático para estas linguagens, deve a princípio, cooperar com o compilador visto que dificilmente tais compiladores seriam modificados para suportar exigências de coleta de lixo. É importante notar que ao se trabalhar com linguagens imperativas e orientadas por objetos o coletor de lixo não deve alterar o valor das palavras endereçadas a não ser que esteja seguro de tais ações. Isto significa que, sem uma análise de *alias* mais detalhada, os objetos não podem ser movidos de suas posições no *heap* aleatoriamente.

Bartlett [Bartlett 1989a] generaliza *copying collection* para desenvolver um coletor de lixo para C e posteriormente C++ [Bartlett 1989b] evitando tal problema: o coletor manipula duas gerações de entidades.

Blocos em uma nova geração são identificados por números pares enquanto os mais antigos por números ímpares. Uma coleta ocorre quando 50 por cento do espaço livre do *heap* já tiver sido requisitado. Objetos de uma geração mais nova são promovidos, em massa, para uma geração mais antiga. Esta geração é examinada e se uma fração maior que 85 por cento do *heap* estiver ocupado, uma coleta *mark and sweep* é efetuada liberando possíveis fragmentos. Em uma fase final os ponteiros são corrigidos passando a referenciar a nova área de memória: a geração antiga é percorrida e os ponteiros para os objetos deslocados no *heap* são substituídos pelos novos endereços.

5 Conclusão

As técnicas mostradas anteriormente abordaram métodos de coleta de lixo que visam minimizar o custo de espaço e tempo gastos na devolução de memória alocada. Neste trabalho não foi tratado o problema de coleta de lixo para sistemas de tempo real. Extensões do algoritmo de *Reference Count* que permitem a coleta de estruturas circulares podem ser encontrados em [Bobrow, 1980]. Um esquema completo de *Copying Collection* pode ser encontrado em [Baker,

1978]. Ao leitor interessado uma referência completa sobre o assunto pode ser encontrada em [Jones, 1996].

Referências

[1] Appel, A.W. 1997. *Modern Compiler Implementation in ML.* Cambridge University Press.

[2] Baker, H. 1978. *List processing in Real Time on a Serial Computer.* Communications of the ACM. Vol. 21, no. 4, pp 280-294.

[3] Bartlett, J.F. 1989. *Scheme-C: A Portable Scheme-to-C compiler.* Techinical report, DEC Western Research Laboratory, Palo Alto, CA, January 1989.

[4] Bartlett, J.F. 1989. *Mostly-Copying Garbage Collection Picks up Generations and C++.* Techinical Note, DEC Western Research Laboratory, Palo Alto, CA, October 1989.

[5] Bobrow, D.G. 1980. *Managing reentrant structures using reference-counts.* ACM TOPLAS. Vol. 2, no.3, pp. 269-73.

[6] Dijkstra, E.W., Lamport, L. 1978 *On the fly garbage collection - an exercise in cooperation.* Communications of the ACM. Vol. 21, no 11, pp 966-75.

[7] Jones, S.L.P. 1987. *The Implementation of Functional Programming Languages.* C.A.R. Hoare Series Editor.

[8] Jones, R., Lins, R. 1996. *Garbage Collection.* Jonh Wiley Sons Ltd, Baffins Lane, Chichester.

[9] Knuth, D. 1976. *The Art of Computer Programming.* Vol 1, Section 2.5. Addison Wesley Publishing Company.

9 Coleta de Lixo

Compilação e Otimização de Código Machina para Código C

Fábio Tirelo

Compilação e Otimização de Código Machĩna para Código C*

Fabio Tirelo
ftirelo@dcc.ufmg.br

3 de julho de 1999

Resumo

Neste artigo, abordamos o problema de compilação e otimização de código escrito na linguagem Machĩna para código C. Machĩna é uma linguagem baseada em Máquinas de Estado Abstratas e possui diversas construções de alto nível. A maior dificuldade na compilação de código Machĩna para C está na tradução do modelo de execução de Machĩna, que é essencialmente paralelo, para o modelo seqüencial de C. Duas fontes de otimização são o escalonamento de código com o objetivo de diminuir o número de temporários, e a dedução de desvios, para evitar a avaliação desnecessária de guardas.

1 Introdução

1.1 Máquinas de Estado Abstratas

Máquinas de Estado Abstratas (ASM) é um modelo de especificação, no qual sistemas são definidos operacionalmente, por meio das mudanças de estado causadas por sua regra de transição. Neste modelo, existe um estado atual, e um novo estado é criado executando-se a regra de transição neste estado.

Uma regra de transição tem a forma de um programa de uma linguagem imperativa como Pascal, e pode ser de três tipos principais:

- *atualização* – é da forma $f(\bar{x}) := y$ e tem o efeito de modificar a função f no ponto $\bar{x}$ para retornar o valor y no próximo estado;
- *condicional* – tem a forma **if** g **then** R_1 **else** R_2, onde g é uma expressão booleana e R_1 e R_2, são regras, e tem o efeito de executar R_1, se a guarda g avaliar em *true*, ou executar R_2, caso contrário;
- *bloco* – tem a forma R_1, R_2, onde R_1 e R_2 são regras, e tem o efeito de executar as regras R_1 e R_2 em paralelo.

A execução de um programa ASM consiste na execução da regra de transição *repetidamente*, sendo que o efeito da execução de uma atualização em um passo só é percebido no passo seguinte.

É importante ressaltar que o fluxo de controle é diferente do modelo tradicional de linguagens imperativas. Primeiro, instruções de um mesmo bloco são executadas em paralelo. Desta forma, uma seqüência da forma $x := y, y := x$ tem o efeito, em ASM, de permutar os valores de x e y, enquanto em uma linguagem imperativa comum, tem o efeito de atribuir o valor inicial de y a x e a y. A segunda diferença é que, por restrições do modelo, não existem uma construção lingüística semelhante a um *while* e nem recursividade. Estas restrições têm por objetivo facilitar a demonstração de propriedades do sistema, por tornar óbvio o fluxo de controle. Um comando da

*Seminário apresentado para a disciplina *Tópicos em Compiladores*, do DCC/UFMG.

forma **while** E **do** C de uma linguagem imperativa é simulado em ASM por uma regra condicional da forma **if** E **then** C. Com efeito, esta regra será executada repetidamente e, enquanto a guarda E for verdadeira, a máquina executará a regra C.

1.2 O Escopo do Trabalho

O objetivo principal do trabalho é, criar uma ferramenta de laboratório, com a qual possamos estudar novos aspectos do paradigma de programação de ASM e desenvolver novas técnicas de otimização de código baseado em ASM. Para isso, criamos a linguagem Machĭna, que é baseada em Máquinas de Estado Abstratas. A definição da linguagem pode ser encontrada em [4] e os seus principais recursos são os seguintes:

- suporte à modularidade;
- sistema de tipos fortemente tipado;
- mecanismos para definição de tipos estruturados;
- suporte ao modelo de computação distribuída de ASM, definido em [2].

Com a compilação de Machĭna para C, pretendemos obter eficiência na execução das regras de transição, principalmente para o modelo seqüencial. Além disso, compilação para C permite a inserção de módulos ASM em sistemas já existentes.

As vantagens principais de se escrever módulos de programa em uma linguagem de especificação formal e compilá-los para C, ao invés de escrevê-los diretamente em C são a precisão dos modelos formais e a facilidade de demonstração de propriedades e uso de provadores automáticos de teoremas.

Ao contrário de muito métodos formais, ASM é baseado em modelos matemáticos simples e bem conhecidos: conjuntos e sistemas de transição. Além disso, a demonstração de propriedades é simples, pois o fluxo de controle é trivial, sendo fácil idenficar os passos do algoritmo, e não existem efeitos colaterais na execução de regras e na avaliação de expressões. Mais ainda, é fácil escrever código em ASM, pois a sua forma é de pseudo-código, que é de conhecimento geral. Outros métodos, como a Semântica Denotacional, utilizam conceitos menos usuais, o que pode tornar a leitura e a escrita tarefas muito complicadas.

1.3 Estrutura do Artigo

Este texto está dividido da seguinte maneira:

- Na Seção 2, mostraremos o esquema de compilação separada de módulos de Machĭna;
- Na Seção 3, mostraremos a compilação de declarações de tipos, funções e ações;
- Na Seção 4, mostraremos a compilação de expressões;
- Na Seção 5, mostraremos a compilação de regras de transição;
- Na Seção 6, mostraremos algumas estratégias de otimização do código gerado;
- Na Seção 7, apresentaremos algumas conclusões e os trabalhos futuros.

2 Compilação de Módulos

2.1 Módulos em Machĭna

Um módulo em Machĭna é uma unidade de compilação separada e tem a forma:

```
module X;          module Y;
public f;          import X;
g;                 ...
...                ... f ...
end;               end;
```

Figura 1: Exemplo de Visibilidade

module `module-name`
  importações
  declarações
  inicializações
  regra de transição
end

Um módulo consiste nas seguintes seções:

- *importações* – torna visíveis ao módulo os identificadores que foram definidos nos módulos importados;
- *inicializações* – inicializações especiais de algumas funções;
- *declarações* – podem ser:
  - *tipos* – definição de tipos;
  - *funções* – definição de funções estáticas, dinâmicas, derivadas e externas;
  - *ações* – definição de abstrações de regras de transição;
- *regra de transição* – definição da regra de transição do módulo.

Machĭna possui mecanismos de controle de visibilidade semelhantes a Java. Dentro de um módulo podemos definir elementos públicos ou privados. Um elemento será público, se a sua declaração contiver o modificador de visibilidade **public**. Caso contrário, ele será privado ao módulo. Quando o elemento for público, ele será visível a partir de qualquer módulo que importar o módulo onde foi definido. Caso contrário, ele será visível somente dentro do módulo onde foi definido.

Por exemplo, nos módulos `X` e `Y` da Figura 1, qualquer função definida em `Y` pode acessar a função `f`, definida em `X`, ao passo que a função `g` só pode ser referenciada dentro de `X`.

2.2 Estrutura do Código Gerado Para um Módulo

Implementaremos um esquema de compilação separada, isto é, cada módulo será compilado de maneira independente, sendo necessário somente que os módulos de que depende estejam compilados.

A compilação de um arquivo de nome `X.mach` gera os seguintes arquivos:

- `module_X.c`, que contém a implementação das funções definidas em `X.mach`;
- `module_X.h`, que contém a interface de exportação de funções e a definição de tipos exportados;

Um módulo Machĭna da forma dada na Seção 2.1 é compilado para C da seguinte maneira:

```
/* *** module_X.c *** */
    declaracoes
    funcao de inicializacao:
```

```
    void init_X()
    {
        ...
    }
    funcao de regra de transicao:
    void rule_X()
    {
        ...
    }

/* *** module_X.h *** */
    includes dos arquivos importados
    declaracao dos cabecalhos das funcoes, tipos e acoes em C
```

Para executar a regra de transição de um módulo `X`, deve-se antes executar a sua regra `init`, por meio da função `init_X()` e, em seguida, executa-se a função `rule_X()`.

2.3 Compilação dos Mecanismos de Visibilidade

As informações de visibilidade são representadas no arquivo compilado da seguinte maneira:

- se um módulo `Y` importar um módulo `X`, então, no arquivo `module_Y.h`, haverá a linha

  ```
  #include "module_X.h"
  ```

- se um identificador `f` for declarado como público no módulo `X`, então haverá em `module_X.h` a declaração

  ```
  extern tipo-de-f f;
  ```

  de modo que `f` seja visível a qualquer módulo que importar `X`.

Por exemplo, na compilação dos módulos da Figura 1, são gerados os arquivos de cabeçalho C abaixo, que contêm as informações de visibilidade e exportações:

```
/* Arquivo module_X.h */
#ifndef __module_X_h
#define __module_X_h

extern int f; /* Torna f visível a qualquer modulo que importar X */

#endif

/* Arquivo module_Y.h */
#ifndef __module_Y_h
#define __module_Y_h

#include "module_X.h" /* Importa os elementos publicos de X */

#endif
```

3 Compilação de Declarações

Em Machina, há três tipos de declarações: de Tipos, de Funções e de Ações. Nesta seção mostraremos o código gerado para cada um destes tipos de declarações.

Bool	char
Char	char
Int	int
Real	double
Enum	char
String	char *

Tabela 1: Compilação dos Tipos Básicos e Enumeração.

3.1 Compilação de Declarações de Tipos

Um tipo em Machĭna pode ser um dos seguintes:

- Tipos Básicos – `Bool`, `Char`, `Int`, `Real` e `String`
- Tipos Derivados – enumeração e intervalo;
- Tipos Estruturados – tuplas, listas e conjuntos.

Os tipos básicos e a enumeração são compilados para os tipos da Tabela 1.

Um tipo *sub-range*, S, é compilado da seguinte maneira: são geradas três constantes, `minS`, `maxS` e `sizeS`, que representam, respectivamente, os limites inferior e superior de S e o número de elementos do intervalo; qualquer variável do tipo S é declarada em C como do tipo do qual S é intervalo. Por exemplo, considere o código Machĭna abaixo:

```
type T is 1..1000;
dynamic x : T := 10;
```

Este código é compilado para C como

```
#define minT 1
#define maxT 1000
/* sizeT = maxT - minT + 1 */
#define sizeT 1000
int x = 10;
```

A compilação de tuplas rotuladas gera uma estrutura de C, na qual cada campo possui o tipo do elemento correspondente. Por exemplo, considere a compilação do tipo tupla definido por:

```
type T is (a : Int, b : Real, c : Char);
```

O código gerado é dado por:

```
typedef struct tupleT
{
    int a;
    double b;
    char c;
} * T;
```

A compilação de união disjunta gera uma estrutura que contém dois elementos: um *flag* para determinação do tipo do elemento e uma união disjunta de C. Por exemplo, considere a compilação de:

```
type T is Int | Real | Char;
```

O código gerado é dado por:

```
typedef struct unionT
{
    int flagT;
    union
    {
        int a;
        double b;
        char c;
    } theVal;
} * T;
```

Para compilação de listas, necessitamos das seguintes operações: obtenção do primeiro elemento (*head*), obtenção da cauda da lista (*tail*), concatenação de listas (*append*), comprimento (*lenght*) e construção de uma nova lista (*cons*).

Para estas operações, necessitamos de uma estrutura de lista encadeada. A declaração do tipo `machina_list` em C é dada por:

```
typedef struct mach_list
{
    void * info;
    struct mach_list * next;
} * machina_list;

mach_list * append(mach_list L1, mach_list L2);
int length(mach_list L);
mach_list * cons(void * e, mach_list * L);
```

Para a compilação de conjuntos, necessitamos das seguintes operações: conjunto vazio, inserção de elementos no conjunto, união, interseção, diferença, pertinência e comparação.

Dependendo do tipo dos elementos do conjunto, podemos escolher entre duas abordagens: vetores de *bits* e tabela *hash*. Vetores de *bits* podem ser utilizados para os casos em que o número máximo de elementos é pequeno, como por exemplo, para `Char`, `Bool`, enumerações e intervalos pequenos de inteiros. Nos outros casos, utiliza-se tabela *hash*.

As operações para a primeira abordagem são definidas como:

Operação	**Implementação**
emptySet()	$\langle 0, 0, \cdots, 0 \rangle$
newSet(x)	`(1 << x)`
newSet(x,y)	`((1 << (y+1)) - (1 << (x+1))`
insertInSet(x,s)	`s | newSet(x)`
union(A,B)	`A | B`
intersection(A,B)	`A & B`
diff(A,B)	`A & !B`
inSet(x,C)	`newSet(x) & C`
le(A,B)	`diff(A,B) == emptySet()`
ge(A,B)	`le(B,A)`
equals(A,B)	`le(A,B) && le(B,A)`

Para a segunda abordagem, definimos as operações:

Operação	Implementação
emptySet()	Hash vazia
newSet(x)	`h = emptySet(); insere(x,h)`

newSet(x,y)	`h = emptySet();`
	`for (i = x; i <= y; i++)`
	`  insere(i,hash);`
	`return h;`

Union(A,B)	`H = clone(A);`
	`for (i = 0; i < tam(B); i++)`
	`  if (lookup(B[i], A) == NULL)`
	`  insere(B[i], H);`
	`  return H;`

Intersection(A,B)	`H = emptySet();`
	`for (i = 0; i < tam(A); i++)`
	`  if (lookup(A[i], B) != NULL)`
	`  insere(A[i], H);`
	`return H;`

Diff(A,B)	`H = emptySet();`
	`for (i = 0; i < tam(A); i++)`
	`  if (lookup(A[i], B) == NULL)`
	`  insere (A[i], H);`
	`return H;`

InSet(x,C)	`lookup(x,C) != NULL`

le(A,B)	`for (i = 0; i < tam(A); i++)`
	`  if (lookup(A[i], B) == NULL)`
	`  return false;`
	`return true;`

ge(A,B)	`LE(B,A)`
equals(A,B)	`LE(A,B) && LE(B,A)`

3.2 Compilação de Declarações de Funções

As funções que possuem corpo para ser avaliado são compiladas para funções em C. Por exemplo, a declaração de função:

```
f(x : Int) : Int := x + 1
```

é compilada para C como a função:

```
int f(int x) { return x + 1; }
```

Como a forma geral de uma função com corpo é:

```
f(parametros) : tipo := expressao
```

a compilação de uma função com corpo é:

```
tipo f(parametros) { return expressao; }
```

A compilação de expressões será mostrada na Seção 4.

Supondo que f seja uma função $f : A \to B$, estática ou dinâmica, dada por um mapeamento, isto é, que não possui um corpo para ser avaliado, então, f é compilada da seguinte maneira:

- se A for `Bool`, `Char`, enumeração ou algum intervalo cujo número de elementos é menor que $2^{16} = 65536$, então f é declarada como:

  ```
  B f[num-elementos-A];
  ```

onde `num-elementos-A` é o número de elementos do tipo A;

- caso contrário, define-se uma estrutura `mappingAB` da forma:

```
typedef struct
{
    A inVal;
    B outVal;
} * mappingAB;
```

e declara, em C, a função f como:

```
machina_mapping f;
```

onde `machina_mapping` é uma estrutura de tabela *hash*, cuja chave de busca é um elemento do tipo `A`. É gerada também uma função `hashA : A` $\rightarrow$ `int`, que calcula a função *hash* de um elemento do tipo `A`, para consulta à tabela `f`. A consulta a `f` será mostrada na Seção 4.4.

Por exemplo, considere o código Machĩna abaixo:

```
type SubRange is 1..100;
     OutroSubRange is 1..100000;
     Tupla is (a:Int, b:Int);
     OutraTupla is (s1:SubRange, s2:SubRange);
     MaisUmaTupla is (s1:SubRange, s2:OutroSubRange);
dynamic a : Int -> Int;
        b : Tupla -> Int;
        c : OutraTupla -> List;
        d : MaisUmaTupla -> MaisUmaTupla;
```

A compilação deste código para C é dado por:

```
int sizeSubRange = 100;
int sizeOutroSubRange = 100000;
typedef struct tupleTupla
{ int a; int b; } * Tupla;
typedef struct tupleOutraTupla
{ int s1; int s2; } * OutraTupla;
typedef struct tupleMaisUmaTupla
{ int s1; int s2; } * MaisUmaTupla;
machina_mapping a;
machina_mapping b;
List c[tamSubRange][tamSubRange];
typedef struct
{ MaisUmaTupla inVal; MaisUmaTupla outVal; } * mappingMUTMUT;
machina_mapping d;
/* Calculo da funcao hash para valores do tipo MaisUmaTupla */
int hashMaisUmaTupla(MaisUmaTupla inVal)
{
    ...
}
```

3.3 Compilação de Declarações de Ações

Uma declaração de ações em Machĭna tem a forma:

```
action a(parametros) := regra end
```

A compilação de uma ação gera uma função em C da forma:

```
void action_a(parametros)
{
    Recolhe as atualizacoes da regra
    return;
}
```

As atualizações de uma ação são armazenadas em uma lista de atualizações (*update-list*), e serão disparadas ao final da execução da transição, como mostrado na Seção 5. Na Seção 6.3, mostraremos os casos em que não é necessário salvar atualizações para posterior disparo.

Por exemplo, a ação

```
action a(x : Int) :=
    f(a,b) := 1, g(x) := 2
end
```

é compilada para:

```
void action_a(int x)
{
    save_update("address of f(a,b)", 1);
    save_update("address of g(x)", 2);
    return;
}
```

4 Compilação de Expressões

Nesta seção descreveremos o esquema de compilação das expressões de Machĭna para expressões em C. A compilação de uma expressão E_M, em Machĭna, para uma expressão equivalente E_C, em C, é representada por $\mathcal{E}[\![E_M]\!] = E_C$. As funções $\mathcal{O}$ e $\mathcal{B}$ são utilizadas para tradução de operadores e literais, respectivamente. Por exemplo, $\mathcal{B}[\![1]\!] = 1$. A diferença entre o número **1** do lado esquerdo e o do lado direito é que o primeiro é o literal em Machĭna e o segundo em C. O objetivo de utilizar estas funções é fazer adaptações quando os literais e os operadores forem diferentes entre as duas linguagens, como, por exemplo, o operador `=` de Machĭna que é compilado para o operador `==` de C.

Para permitir maior flexibilidade nas otimizações, o código gerado será semelhante a quádruplas, mas em um nível mais alto, permitindo inclusive chamadas de funções em expressões.

4.1 As Expressões de Machĭna

As expressões de Machĭna podem ser divididas entre as seguintes categorias:

- expressões lógicas e aritméticas;
- operações sobre tipos estruturados – listas, tuplas, união disjunta e conjuntos;
- chamadas de funções;

4.2 Expressões Lógicas e Aritméticas

4.2.1 Expressões Booleanas

As expressões booleanas de Machĩna são os literais `true` e `false`, a operação unária `not` e as operações binárias `and` e `or`. Para as três primeiras, a compilação é dada por:

Machĩna		**C**
$\mathcal{E}[\![\texttt{true}]\!]$	=	`1`
$\mathcal{E}[\![\texttt{false}]\!]$	=	`0`
$\mathcal{E}[\![\textbf{not E}]\!]$	=	`!` $\mathcal{E}[\![E]\!]$

A semântica de Machĩna define que a avaliação de expressões booleanas possui curto-circuito. Isto significa que, na avaliação de E_1 **and** E_2, E_2 só será avaliada se E_1 for verdadeiro. Da mesma forma, na avaliação de E_1 **or** E_2, E_2 só será avaliada se E_1 for falso.

Desta maneira, o código gerado para as expressões **and** e **or** será da forma:

Expressão	**Código**

E_1 **and** E_2	`if (!` $\mathcal{E}[\![E_1]\!]$`) goto L1;`
	`if (!` $\mathcal{E}[\![E_2]\!]$`) goto L1;`
	`/* Código para` E_1 **and** E_2 `verdadeiro */`
	`...`
	`goto L2;`
	`/* Código para` E_1 **and** E_2 `falso */`
	`L1: ...`
	`/* Código independente de` E_1 **and** E_2 `*/`
	`L2: ...`

E_1 **or** E_2	`if (`$\mathcal{E}[\![E_1]\!]$`) goto L1;`
	`if (!` $\mathcal{E}[\![E_2]\!]$`) goto L2;`
	`L1: /* Código para` E_1 **or** E_2 `verdadeiro */`
	`...`
	`goto L3;`
	`/* Código para` E_1 **or** E_2 `falso */`
	`L2: ...`
	`/* Código independente de` E_1 **or** E_2 `*/`
	`L3: ...`

4.2.2 Caracteres

As expressões caracteres são simplesmente os literais e as funções `chr(n)`, que retorna o `n`-ésimo caractere da tabela ASCII, e `ord(c)`, que retorna o índice, na tabela ASCII, de `c`. A compilação destas expressões é:

Machĩna		**C**
$\mathcal{E}[\![c]\!]$	=	$\mathcal{B}[\![c]\!]$
$\mathcal{E}[\![\text{chr}(E)]\!]$	=	`(char)(`$\mathcal{E}[\![E]\!]$`)`
$\mathcal{E}[\![\text{ord}(E)]\!]$	=	$\mathcal{E}[\![E]\!]$ `- '0'`

4.2.3 Números Inteiros

As expressões sobre os números inteiros são os literais, operações aritméticas unárias e binárias e as funções pré-definidas `max`, `min`, `abs` e `sqr`. A compilação destas expressões é:

Machĩna		C
$\mathcal{E}[\![n]\!]$	=	$\mathcal{B}[\![n]\!]$
$\mathcal{E}[\![E_1 \text{ op } E_2]\!]$	=	$\mathcal{E}[\![E_1]\!]$ $\mathcal{O}[\![\texttt{op}]\!]$ $\mathcal{E}[\![E_2]\!]$
$\mathcal{E}[\![\text{op } E]\!]$	=	$\mathcal{O}[\![\texttt{op}]\!]$ $\mathcal{E}[\![E]\!]$
$\mathcal{E}[\![\max(E_1, E_2)]\!]$	=	`(t1 =` $\mathcal{E}[\![E_1]\!]$`, t2 =` $\mathcal{E}[\![E_2]\!]$`, (t1>t2)?t1:t2)`
$\mathcal{E}[\![\min(E_1, E_2)]\!]$	=	`(t1 =` $\mathcal{E}[\![E_1]\!]$`, t2 =` $\mathcal{E}[\![E_2]\!]$`, (t1<t2)?t1:t2)`
$\mathcal{E}[\![\text{abs}(E)]\!]$	=	`(t =` $\mathcal{E}[\![E]\!]$`, (t<0)?-t:t)`
$\mathcal{E}[\![\text{sqr}(E)]\!]$	=	`(t =` $\mathcal{E}[\![E]\!]$`, t*t)`

4.2.4 Números Reais

As expressões sobre os números reais são as mesmas sobre os inteiros, com exceção da operação de resto da divisão, além de haver a operação de raiz quadrada, dada por `sqrt(x)`. A compilação destas expressões é:

Machĩna		C
$\mathcal{E}[\![n]\!]$	=	$\mathcal{B}[\![n]\!]$
$\mathcal{E}[\![E_1 \text{ op } E_2]\!]$	=	$\mathcal{E}[\![E_1]\!]$ $\mathcal{O}[\![\texttt{op}]\!]$ $\mathcal{E}[\![E_2]\!]$
$\mathcal{E}[\![\text{op } E]\!]$	=	$\mathcal{O}[\![\texttt{op}]\!]$ $\mathcal{E}[\![E]\!]$
$\mathcal{E}[\![\max(E_1, E_2)]\!]$	=	`(t1 =` $\mathcal{E}[\![E_1]\!]$`, t2 =` $\mathcal{E}[\![E_2]\!]$`, (t1>t2)?t1:t2)`
$\mathcal{E}[\![\min(E_1, E_2)]\!]$	=	`(t1 =` $\mathcal{E}[\![E_1]\!]$`, t2 =` $\mathcal{E}[\![E_2]\!]$`, (t1<t2)?t1:t2)`
$\mathcal{E}[\![\text{abs}(E)]\!]$	=	`(t =` $\mathcal{E}[\![E]\!]$`, (t<0.0)?-t:t)`
$\mathcal{E}[\![\text{sqr}(E)]\!]$	=	`(t =` $\mathcal{E}[\![E]\!]$`, t*t)`
$\mathcal{E}[\![\text{sqrt}(E)]\!]$	=	`sqrt(`$\mathcal{E}[\![E]\!]$`)`

4.2.5 Cadeias de Caracteres

As operações sobre cadeias de caracteres são os literais, a concatenação pelo operador `+` e o comprimento. A compilação destas expressões é:

Machĩna		C
$\mathcal{E}[\![s]\!]$	=	$\mathcal{B}[\![s]\!]$
$\mathcal{E}[\![E_1 + E_2]\!]$	=	`concat(`$\mathcal{E}[\![E_1]\!]$`,` $\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![\text{lenght}(E)]\!]$	=	`strlen(`$\mathcal{E}[\![E]\!]$`)`

A função `concat` aloca uma nova cadeia de caracteres e chama a função `strcat` de C, retornando o resultado nesta nova cadeia.

4.3 Expressões sobre Tipos Estruturados

4.3.1 Tuplas

As expressões envolvendo tuplas são os agregados tupla e a seleção de um elemento de uma tupla rotulada, por meio do operador ".". Ao mostrar a compilação, estamos supondo que o tipo T é uma tupla rotulada, contendo n campos, de rótulos f_1, ..., f_n. A compilação das expressões envolvendo tuplas é:

Machĩna		C
$\mathcal{E}[\![()]\!]$	=	`NULL`
$\mathcal{E}[\![T(E_1, ..., E_n)]\!]$	=	`(t.`f_1 `=` $\mathcal{E}[\![E_1]\!]$`, ..., t.`f_n `=` $\mathcal{E}[\![E_n]\!]$`, t)`
$\mathcal{E}[\![E_1.E_2]\!]$	=	`(t =` $\mathcal{E}[\![E_1]\!]$`, (t!=NULL)?(`$\mathcal{E}[\![E_1]\!]$`->`$\mathcal{E}[\![E_n]\!]$`):(NULL))`

4.3.2 Listas

Como mostrado na Seção 3.1, qualquer tipo lista é traduzido como uma lista genérica, cujos elementos são do tipo `void*`, que utiliza as funções definidas sobre as listas. As funções sobre listas são traduzidas da seguinte forma:

Machīna		**C**
$\mathcal{E}[\![\textbf{nil}]\!]$	=	`NULL`
$\mathcal{E}[\![[\,]]\!]$	=	`NULL`
$\mathcal{E}[\![E_1 :: E_2]\!]$	=	`cons(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 @ E_2]\!]$	=	`append(clone(`$\mathcal{E}[\![E_1]\!]$`), clone(`$\mathcal{E}[\![E_2]\!]$`))`
$\mathcal{E}[\![[E_1, ..., E_n]]\!]$	=	`cons(`$\mathcal{E}[\![E_1]\!]$`, cons(..., cons(`$\mathcal{E}[\![E_n]\!]$`, NULL)))`
$\mathcal{E}[\![\text{lenght}(E)]\!]$	=	`lenght(`$\mathcal{E}[\![E]\!]$`)`
$\mathcal{E}[\![\text{head}(E)]\!]$	=	$\mathcal{E}[\![E]\!]$`.info`
$\mathcal{E}[\![\text{tail}(E)]\!]$	=	$\mathcal{E}[\![E]\!]$`.next`

4.3.3 Conjuntos

Como mostrado na Seção 3.1, conjuntos podem ser traduzidos para vetores de *bits* ou para tabelas *hash*. Para cada uma destas abordagens, são implementadas as operações necessárias. As expressões de conjuntos utilizam estas operações na compilação, conforme mostrado abaixo:

Machīna		**C**
$\mathcal{E}[\![\{\}]\!]$	=	`emptySet()`
$\mathcal{E}[\![\{E_1, ..., E_n\}]\!]$	=	`union(newSet(`$\mathcal{E}[\![E_1]\!]$`), union(..., newSet(`$\mathcal{E}[\![E_n]\!]$`)))`
$\mathcal{E}[\![E_1 + E_2]\!]$	=	`union(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 - E_2]\!]$	=	`difference(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 * E_2]\!]$	=	`intersection(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 \textbf{ in } E_2]\!]$	=	`inSet(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1(E_2)]\!]$	=	`inSet(`$\mathcal{E}[\![E_2]\!]$`, `$\mathcal{E}[\![E_1]\!]$`)`
$\mathcal{E}[\![E_1 = E_2]\!]$	=	`equals(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 != E_2]\!]$	=	`! equals(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 < E_2]\!]$	=	`! ge(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 > E_2]\!]$	=	`! le(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 <= E_2]\!]$	=	`le(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1 >= E_2]\!]$	=	`ge(`$\mathcal{E}[\![E_1]\!]$`, `$\mathcal{E}[\![E_2]\!]$`)`
$\mathcal{E}[\![E_1..E_2]\!]$	=	`newSet(`$\mathcal{E}[\![E_1]\!]$`,`$\mathcal{E}[\![E_2]\!]$`)`

4.4 Chamadas de Funções

Para uma função com corpo, gera-se simplesmente uma chamada à função C correspondente.

O código gerado para chamadas de funções sem corpo depende do domínio da função. Se o domínio for tal que a função é armazenada em um arranjo, o código gerado é simplesmente uma consulta à posição do elemento desejado no arranjo. Caso contrário, é gerada uma consulta à tabela *hash* relativa à função.

Por exemplo, considere as funções definidas na Seção 3.2 e considere as seguintes chamadas:

```
a(1) /* Chamada 1 */
b(1,2) /* Chamada 2 */
c(3,3) /* Chamada 3 */
d(1,2) /* Chamada 4 */
```

O código C gerado será:

```
(t = hashInt(1), lookup(a,t)) /* Chamada 1 */
(t = hashTupla(1,2), lookup(b,t)) /* Chamada 2 */
c[3][3] /* Chamada 3 */
(t = hashMaisUmaTupla(1,2), lookup(d,t)) /* Chamada 4 */
```

A função de `lookup` retorna um ponteiro para o endereço do elemento, para que o mesmo código possa ser utilizado tanto para o lado direito quanto para o lado esquerdo de atualizações. Quando a chamada for simplesmente de consulta, ele deve ser derreferenciado por meio do operador `*` de C.

5 Compilação de Regras de Transição

Nesta seção mostraremos a compilação de regras de transição de Machĭna. O programa principal de um módulo é dado por uma regra de transição, assim como todas as ações.

Basicamente, uma regra de transição é um conjunto de atualização de funções, com estruturas condicionais, de escolha não-determinista e de paralelismo síncrono.

As regras de atualização e condicionais (ou guardadas) já foram apresentadas na Seção 1. Neste texto, mostraremos somente a tradução de regras de atualização, blocos e regras guardadas, porque estas são as regras mais importantes.

A compilação da seção de transição de um módulo da forma:

```
transition:
    regra de transicao
```

gera a seguinte função em C:

```
void rule_X()
{
    Compilacao da regra de transicao
}
```

A compilação da regra de transição propriamente dita gera um seqüência de desvios condicionais e comandos para salvar as atualizações. No final, há um comando para realizar o disparo das atualizações, que é a atualização efetiva das funções modificadas. Este último comando é representado pela função `fire_updates()`.

5.1 Compilação de Regras de Atualização e Blocos

A compilação de uma regra de atualização gera um comando de salvamento da atualização, isto é, seu endereço e o valor a ser salvo, em uma lista, denominada *update-list*. Desta forma, um bloco de regras gera uma seqüência de comandos de salvamento. Ao fim da regra, há um comando para percorrer a lista e disparar as atualizações, isto é, modificar as funções para o próximo passo.

Por exemplo, considere o trecho de código abaixo:

```
type T is 0..100;
dynamic a,b : T;
        f : (T,T) -> T;
transition:
    a = 10 * a,
    b = a + 2 * b,
    f(a,b) = a + b
```

O código gerado será da forma:

```
    #define sizeT 101
    int a,b;
    int f[sizeT][sizeT];
    ...
L0: save_update(&a, 10 * x);
    save_update(&b, a + 2 * b);
    save_update(&(f[a][b]), a + b);
    fire_updates();
    goto L0;
```

Este código apresenta uma grande deficiência, que é salvar todas as atualizações para depois dispará-las. Isto poderia ser contornado se modificássemos a ordem das instruções, fazendo a atualização para as próprias posições, ao invés de salvar para depois atualizar efetivamente. Desta forma, o código da regra ficaria o seguinte:

```
L0: f[a][b] = a + b;
    b = a + 2 * b;
    a = 10 * a;
    goto L0;
```

Abordaremos esta técnica de otimização na Seção 6.2.

5.2 Compilação de Regras Guardadas

A compilação de regras guardadas de Machĭna é feita da seguinte maneira: para uma regra de transição da forma

$$\texttt{if } g \texttt{ then } R_1 \texttt{ else } R_2 \texttt{ end}$$

onde g é uma guarda e R_1 e R_2 são regras, geram-se os comandos

```
    if (! g) goto L1
    Codigo para R1
    goto L2
L1: Codigo para R2
L2: goto next_IF
```

onde `next_IF` é o rótulo para o comando que deve ser executado após o término da execução da compilação da regra condicional.

Por exemplo, considere a compilação do seguinte código Machĭna:

```
module X
algebra:
    static n : Int := 20;
    type T is 0..n;
    static f(x : Int) : Int := x + 1;
    dynamic fat : T -> Int;
            a : T := 1;
    init
        fat(0) := 1
transition:
    if (a < n) then
        a := a + 1,
        fat(a) := a * fat(a - 1)
    end
end
```

A compilação deste módulo gera o seguinte código em C:

```
int n = 20;
#define sizeT 21
int f(int x)
{
    return x + 1;
}
int fat[sizeT];
int a = 1;
void init_X()
{
    fat[0] = 1;
}
void rule_X()
{
    if (a >= n) goto L;
    save_update(&a, a + 1);
    save_update(&(fat[a]), a * fat(a - 1));
    fire_updates();
 L: return;
}
```

Observe que o código de `rule_X` pode ser melhorado para:

```
void rule_X()
{
    if (a >= n) goto L;
    fat[a] = a * fat[a - 1];
    a = a + 1;
 L: return;
}
```

6 Estratégias de Otimização

6.1 Otimização de Desvios

Um grande problema da implementação de Machĩna sugerida na Seção 5 é que cada passo consistia em uma chamada de função, a execução do passo e um retorno. Para permitir a otimização do caso seqüencial, que consiste em executar a mesma regra de transição repetidas vezes até o fim, Machĩna diferencia execução intercalada (de mais de um agente) da execução não-intercalada (com somente um agente). A execução é intercalada, se a definição da transição é da forma:

```
interleaved transition:
    regra
```

Caso contrário, considera-se que a execução é não-intercalada.

No exemplo apresentado na Seção 5.2, temos um caso de execução seqüencial, isto é, não-intercalada. Isto significa que o código da regra pode ser otimizado, acrescentando-se, após o código do bloco, um `goto` para o primeiro comando da função, como mostrado abaixo:

```
void rule_X()
{
 I: if (a >= n) goto L;
    fat[a] = a * fat[a - 1];
    a = a + 1;
    goto I;
```

```
 L: return;
}
```

Este código produz o mesmo resultado que o anterior, e é mais eficiente, pois evita os retornos e chamadas de função entre cada passo, substituindo-os por um `goto`.

Um outro caso de interesse na otimização dos desvios é quando o programa consiste em `if`'s aninhados, como em:

```
if g1 then
    if g2 then R1 else R2 end
else
    if g3 then R3 else R4 end
end
```

Em tempo de compilação, sabemos quais caminhos são percorridos até a execução de cada uma das regras `R`i. Da mesma forma, inspecionando-se os lados esquerdos das atualizações das regras, sabemos quais execuções podem modificar o valor das guardas já avaliadas. Com esta informação, é possível determinar quais guardas irão necessariamente produzir o mesmo resultado no próximo passo, direcionando a execução para o mesmo caminho, o que nos permite evitar que tais guardas sejam re-avaliadas.

No exemplo, acima, suponha que `R1` modifique somente a guarda `g2`, `R2` modifique `g1` e `R3` e `R4` modifiquem a guarda `g1`. Isto significa que, após a execução de `R2`, `R3` e `R4`, deve-se desviar para o ponto no código onde a guarda `g1` é avaliada. Entretanto, como `R1` não modifica `g1`, a saída de `R1` não precisa ser para a avaliação de `g1`. Desta maneira, pode haver um desvio de `R1` diretamente para a avaliação de `g2`, como mostrado abaixo:

```
/* Codigo para a regra de transicao acima */
Ag1: if (! g1) goto Ag3;
Ag2: if (! g2) goto Eg2;
     ... /* Execucao de R1 */
     goto Ag2; /* Evita re-avaliacao de g1 */
Eg2: ... /* Execucao de R2 */
     goto Ag1; /* Desvia para avaliacao de g1 */
Ag3: if (! g3) goto Eg3;
     ... /* Execucao de R3 */
     goto Ag1; /* Desvia para avaliacao de g1 */
Eg3: ... /* Execucao de R4 */
     goto Ag1; /* Desvia para avaliacao de g1 */
```

O resultado da otimização pode ser visto no grafo de fluxo da Figura 2.

6.2 Escalonamento de Código

Seja B um bloco composto por $R_1, ..., R_2$, onde cada R_i é uma regra de atualização ou guardada. O problema de escalonamento deste bloco consiste em ordenar as regras R_i, de modo a evitar o uso de temporários ou inserções na lista de atualizações. Por exemplo, suponha que B seja o bloco

$$\mathtt{x} := \mathtt{e1}, \mathtt{f(x)} := \mathtt{e2}$$

A compilação preservando a ordem deveria salvar o valor de `x` para usá-lo no segundo comando, antes da alteração. Entretanto, pode-se modificar a ordem das duas regras para evitar o uso do temporário.

No caso geral, o escalonamento é feito em 3 fases:

1. coleta das informações de usos e atualizações dos blocos;

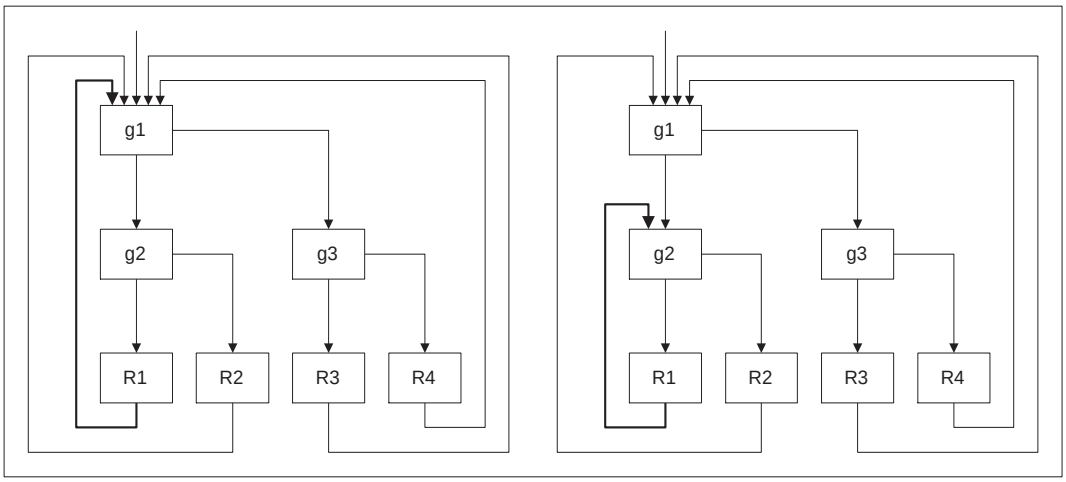


Figura 2: Grafos de Fluxo Sem a Otimização de Desvios e Com a Otimização de Desvios.

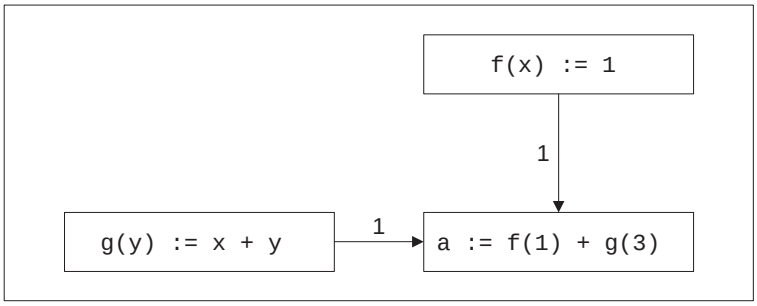


Figura 3: Exemplo de Grafo de Conflitos

2. construção do grafo de conflitos;
3. escalonamento do código.

Na primeira fase, analisa-se o bloco para verificar quais funções são modificadas e quais são consultadas. Por exemplo, para o bloco

$$\mathtt{f(x)} := 1, \mathtt{g(y)} := \mathtt{x} + \mathtt{y}, \mathtt{a} := \mathtt{f(1)} + \mathtt{g(3)}$$

esta fase determina que as funções modificadas e consultadas, para cada regra, são as seguintes:

Regra	Modificadas	Consultadas
`f(x) := 1`	`f`	`x`
`g(y) := x + y`	`g`	`x y`
`a := f(1) + g(3)`	`a`	`f g`

Na segunda fase, constrói-se o grafo de conflitos para o bloco. Os vértices do grafo de conflitos são os componentes do bloco e um arco une um componente R_1 a outro componente R_2, se R_1 modificar alguma função consultada em R_2. A cada arco está associado um peso, que representa o número de conflitos, isto é, o número de funções atualizadas em R_1 que conflitam com R_2. Por exemplo, o grafo de conflitos para o exemplo anterior é mostrado na Figura 3.

A terceira fase é responsável por escalonar, para evitar o máximo possível de salvamento de atualizações. Como este é um problema NP-Completo, devemos utilizar heurísticas para resolvê-lo. Uma heurística que pode ser utilizada é a seguinte:

1. Escalonar os vértices que possuírem grau de saída igual a zero, pois não haverá necessidade de salvamento para tais vértices;

2. Se a primera alternativa falhar, escolhe-se um vértice que possuir a menor diferença entre o grau de saída e o grau de entrada.

Por exemplo, no exemplo anterior, esta heurística escolheria a ordem abaixo, que elimina totalmente a necessidade de salvamentos:

```
a := f(1) + g(3),
f(x) := 1,
g(y) := x + y
```

Um exemplo mais complexo é o escalonamento do algoritmo de ordenação dado por:

```
 1    if modo = 1 and i < n then
 2        k := i,
 3        j := i + 1,
 4        modo := 2
 5    elseif modo = 2 then
 6        if j > n then
 7            modo = 3
 8        else
 9            if f(j) < f(k) then
10                k := j
11            end,
12            j := j + 1
13        end
14    elseif modo = 3 then
15        if k != i then
16            f(k) := f(i),
17            f(i) := f(k)
18        end,
19        i := i + 1,
20        modo := 1
21    end
```

Os blocos que deverão ser analisados para o escalonamento são os blocos B1 (2-4), B2 (9-12), B3 (16-17) e B4 (15-20). O bloco B1 possui 3 componentes, que são os comandos de atualização das linhas 2, 3 e 4. O bloco B2 é formado pelo `if` das linhas 9-11 e pela atualização da linha 12. O bloco B3 é formado pelas atualizações das linhas 16-17. O bloco B4 é formado pelo `if` das linhas 15-18 e pelas atualizações das linhas 19 e 20.

Para os blocos B1 e B2, não há necessidade em trocar a ordem das instruções, pois, na ordem dada, não há necessidade de salvamentos. Para o bloco B3, temos que a instrução 16 modifica a função `f` utilizada na instrução 17 e vice-versa. Desta forma, o algoritmo deve ordenar o salvamento da atualização da instrução 16, para evitar conflitos com a instrução 17. Com isso, não há a necessidade de modificar a ordem no bloco B4. Assim, será necessário fazer somente um salvamento em todo o programa.

6.3 Otimização de Ações

Assim como na regra de transição, uma otimização que poderia ser feita é a substituição de inserções na lista de atualizações por atribuições diretas à função. Entretanto, em uma ação parametrizada, pode haver a criação de *aliases*, o que dificulta fazer tal otimização. Uma alternativa

para isso é o *inlining* do código da ação, o que facilita a análise e permite otimizações mais agressivas. No entanto, o *inlining* nem sempre melhora o tempo de execução. Por exemplo, por aumentar demais o tamanho do código, pode causar mais falhas na *cache* de instruções.

Uma ação que não é pública pode ser otimizada a partir da análise de suas chamadas. Esta análise deve dizer quais atualizações podem ser feitas diretamente e quais devem ser inseridas na lista de atualizações. No entanto, se a ação for pública, muitas otimizações não podem ser feitas, pois não haverá informações do uso da ação. Entretanto, atualizações de funções privadas ao módulo podem ser otimizadas.

Por exemplo, considere o trecho de código abaixo:

```
algebra:
    action a(x,y : Int) :=
        if f(x) = g(y) then
            f(x + y) := x + y
        else
            g(x + y) := x - y
        end
    end
transition:
    a(m,n),
    q := f(q)
end
```

Neste caso, temos duas alternativas: *inlining* do código da ação ou geração de uma função para `a` e a chamada na regra de transição do módulo. No primeiro caso, o código gerado seria da forma

```
void rule_X()
{
    x = m; y = n;
    if (f[x] == g[y])
        f[x + y] = x + y;
    else
        g[x + y] = x - y;
    q = f[q];
}
```

Na segunda abordagem, as duas atualizações deveriam ser colocadas na forma de inserção na lista de atualizações. Uma otimização que pode ser feita é analisar os pontos onde `a()` é chamada, para verificar quais elementos são modificados. Uma análise simples mostraria que a única função modificada por `a()` e consultada na regra é `f()`, assim como a regra não altera nenhuma função consultada em `a()`. Desta maneira, o código de atualização para `g()` não precisa ser compilado para uma inserção na lista de atualizações, como mostrado abaixo:

```
void action_a(int x, int y)
{
    if (f[x] == g[y])
        save_update(&(f[x + y]), x + y);
    else
        g[x + y] = x - y;
}
void rule_X()
{
    action_a(m,n);
    q := f(x);
    fire_updates();
}
```

7 Conclusões

Neste trabalho mostramos o esquema de compilação de módulos de Machĭna e das principais declarações, expressões e regras de transição. Além disso, mostramos algumas otimizações que podem ser feitas no código, de modo a obtermos maior eficiência na execução. A continuação deste trabalho consiste em escrever os esquemas de compilação das outras expressões e regras, assim como estudar e implementar outras otimizações, tais como, *code hoisting*, eliminação de sub-expressões comuns e outras mostradas em [3, 1]. Este estudo consistirá em modificarmos estes algoritmos para não levarem em conta efeitos colaterais na execução de um passo, de modo que eles se apliquem mais efetivamente a programas Machĭna.

Referências

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

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

[3] S.S. Muchnick. *Advanced Comiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[4] F. Tirelo, R.S. Bigonha, M. A. Maia, and V.O. Iorio. Machĭna: A Linguagem de Especificação de ASM. Technical Report 08/1999, Laboratório de Linguagens de Programação, Universidade Federal de Minas Gerais, 1999.

10 Compilação e Otimização de Código Machina para Código C

Processador para o Bytecode de Java

Flávia Peligrinelli Ribeiro

11 Processador para o Bytecode de Java