Chapter 5

Otimização de Código

Otimizar um código é transformá-lo em um código que faz a mesma coisa mas que é mais rápido que a versão anterior. O código transformado pode ser o código fonte da linguagem, o código intermediário gerado pelo compilador, o código em assembler ou mesmo o programa em forma de árvore de sintaxe abstrata.

Emprega-se o termo "otimizar" não só com relação a aumentar a velocidade de execução do código como também diminuir o seu tamanho. Neste curso, a menos de menção em contrário, otimizar terá o sentido de "aumentar a velocidade de execução". O nome "otimizar" significa "tornar o melhor possível". Como apenas em situações excepcionais o código otimizado por um compilador é o melhor que se pode obter, este nome é utilizado incorretamente.

5.1 Blocos Básicos e Grafos de Fluxo de Execução

Um bloco básico é uma seqüência de instruções em assembler que estão em seqüência no código gerado pelo compilador de tal forma que o fluxo de controle se inicia na primeira instrução do bloco básico e termina na última. Nenhuma instrução do bloco básico, exceto a última, pode ser desvio condicional ou incondicional (`goto`, `goto<`, ...). Nenhuma instrução, exceto a primeira, é o alvo de um desvio condicional ou incondicional.

Aho, Sethi e Ullman [4] fornecem um algoritmo para encontrar os blocos básicos de uma seqüência de instruções em assembler, descrito a seguir.

1. Primeiro determinamos os *líderes* do código em assembler. Um líder é a primeira instrução de um bloco básico, e é encontrado pelas regras apresentadas abaixo.

   - A primeira instrução é um líder. Isto é, a instrução onde o código inicia a sua execução é um líder.
   - As instruções que são alvos de desvios são líderes. Estas instruções possuem obrigatoriamente *label* no assembler utilizado.
   - Qualquer instrução que se segue a um desvio é um líder.

2. O bloco básico correspondente a cada líder se inicia nele e termina imediatamente antes do fim do próximo líder ou no fim do programa.

Como exemplo, considere que o programa em S2

```
  var i, j : integer;
begin
i = 0;
while i < 10 do
  begin
  if i > 5
  then
    j = 10;
    while j > 0 do
      j = j - 1;
  else
    j = 5;
  endif
  i = i + 1;
  end
end
```

seja traduzido para

```
( 1)     mov i, 0
( 2)     goto L1
( 3) L2: cmp i, 5
( 4)     goto<= L3
( 5)     mov j, 10
( 6)     goto L4
( 7) L5: sub j, 1
( 8) L4: cmp j, 0
( 9)     goto> L5
(10)     goto L6
(11) L3: mov j, 5
(12) L6: add i, 1
(13) L1: cmp i, 10
(14)     goto< L2
(15)     exit
```

Os blocos básicos deste código são

```
1  2
3  4
5  6
7
8  9
10
11
12
13  14
15
```

Blocos básicos possuem duas características importantes:

Figure 5.1: Grafo do Fluxo de Execução de um programa S2

1. uma vez que o fluxo de execução começa na primeira instrução, ele prossegue até a última sem interrupção por desvios e;

2. não há desvios para o meio de um bloco básico. A execução sempre se inicia na primeira instrução.

Grafos de Fluxo de Execução

Um grafo G = (V, E) é composto por um conjunto de vértices V e arestas E. Uma aresta é um par (v, w) onde v e w são vértices do grafo. Dizemos que v e w estão ligados por uma aresta. Em um *grafo dirigido*, a aresta (w, v) é diferente de (v, w), sendo esta última uma aresta *de* v *para* w.

Um caminho em um grafo é uma seqüência de arestas (v_1, v_2), (v_2, v_3), (v_3, v_4), ... (v_{n-1}, v_n) que ligam v_1 a v_n. Um ciclo é um caminho onde v_1 a v_n.[1]

O fluxo de execução de um programa pode ser visualizado criando-se um grafo dirigido a partir dos seus blocos básicos. Cada vértice do grafo é um bloco básico. Existirá uma aresta de um bloco B1 para um bloco B2 se B2 puder ser executado imediatamente após B1, o que acontecerá se:

1. houver um desvio (`goto`) condicional ou incondicional da última instrução de B1 para a primeira de B2;

2. B2 seguir-se a B1 no programa e B1 não terminar com um `goto` incondicional. Neste caso, existe em algum ponto do programa um desvio para a primeira instrução de B2: foi esta a razão pela qual B2 foi separado de B1.

Um grafo construido de acordo com as regras acima será chamado de *Grafo de Fluxo de Execução*. Para o programa apresentado na seção 5.1, o grafo de fluxo de execução é aquele mostrado na Figura 5.1.

Nas seções seguintes, descrevemos as otimizações mais comuns sem entrar em detalhes sobre os algoritmos necessários para realizá-las.

5.2 Otimizações em Pequena Escala

Otimizações em pequena escala (*peephole optimizations*) possuem este nome porque são feitas examinando-se poucas instruções do código gerado, em geral do assembler.

Para compreender as otimizações *peephole* (e otimizações em geral), devemos ter em mente que:

1. Um desvio incondicional, como `goto>`, `goto<=`, etc, nem sempre causa um desvio. Quando não causa, o seu tempo de execução é menor do que o de um `goto`. Quando causa, o `goto` é mais rápido.

[1] Um grafo dirigido sem ciclos é chamado em Inglês de *direct acyclic graph*, abreviado por DAG.

2. Uma multiplicação ou divisão é muito mais lenta do que uma soma, subtração ou deslocamento de bits.

3. Operadores de manipulação de bits como & e | binários de C++ são muito mais rápidos do que as operações aritméticas.

4. O uso de constantes é mais rápido do que o de variáveis:
```
mov a, 2
```
é mais rápido do que
```
mov a, b
```

Colocaremos o código original seguido do código otimizado dentro de um retângulo *ou* o código original seguido de uma barra horizontal seguido do código otimizado. As otimizações triviais não serão comentadas.

1.
```
mov t, a
mov a, t
```

```
mov t, a
```

Este código pode ter sido o resultado da tradução de mais de uma linha do código fonte original, como
```
t = a;
a = t + 1;
```

2.
```
push a
pop a
```

```
(nada)
```

3.
```
    cmp 5, 3
    goto<> L2
L1: mov a, 1
L2:
```

```
    goto L2
L1: mov a, 1
L2:
```

A comparação será sempre verdadeira e não precisa ser feita.

Embora seja improvável que o programador gere este código, ele pode ser o resultado do uso de constantes no programa, como neste exemplo:

```
#define Max 5
...
if ( Max == 3 )
   a = 1;
```

Admitimos que o registrador `cm` é zerado pela instrução `goto<>`. Se não fosse, o código otimizado produziria um resultado diferente do original por causa deste registrador. Na versão original, o registrador seria inicializado e, não otimizada, não.

4. Eliminação de salto sobre salto.

```
     cmp a, 1
     goto== L1
     goto L2
L1:  add x, y
L2:
```

```
     cmp a, 1
     goto<> L2
L1:  add x, y
L2:
```

5.
```
     goto L1
     ...
L1:  goto L2
     ...
L2:  ...
```

```
     goto L2
     ...
L1:  goto L2
     ...
L2:
```

6.
```
     goto L1
     ...
     goto L4
```

```
L1:  cmp a, b
     goto> L2
L3:  ...
```

```
L1:  cmp a, b
     goto> L2
     goto  L3
     ...
     goto L4
     L3:
```

As duas versões possuem o mesmo número de comandos. Contudo, a última versão é mais rápida porque o `goto L3` só será executado quando a comparação falhar, enquanto que na versão não otimizada, `goto L1` sempre será executado.

7. Simplificações Algébricas.

   Estas otimizações serão apresentadas em C++ por uma questão de clareza.

```
a = b + 0
a = b * 1
a = b * 0
a = b / 1
a = 0 - b
a = b - 0
```

```
a = b
a = b
a = 0
a = b
a = -b
a = b
```

   Instruções como as acima geralmente não são produto da codificação direta do programador, mas o resultado da substituição de constantes:

```
#define Max 1
...
size =  n*Max;
```

8. Transformações Utilizando Operações Dependentes de Máquinas.

```
8*a
```

```
a << 3
```

```
a/16
```

```
a >> 4
```

```
a%2
```

```
a&1
```

```
160*a
```

```
(a << 7) + (a << 5)
```

Esta otimização é obtida por:

```
160*a = 32*5*a = 32*(4 + 1)*a = 128*a + 32*a = (a << 7) + (a << 5)
```

Muitas máquinas possuem uma instrução `inc x` que incrementa `x` de `1`. Esta operação é muito mais rápida, geralmente, do que somar `1` a `x` com "`add x, 1`".

Assim, soma pode ser otimizada:

```
add  x, 1
```

```
 inc x
```

```
add x, 2
```

```
inc x
inc x
```

Em geral é mais rápido chamar `inc` duas vezes do que utilizar `add x, 2`.

5.3 Otimizações Básicas

1. Subexpressões Comuns.

   Uma expressão será chamada de "subexpressão comum" se ela aparecer em dois lugares diferentes e se as suas variáveis não tiverem mudado de valor entre o cálculo de um expressão e outra. Pode-se calcular o valor da subexpressão apenas uma vez. Como exemplo, o codigo:

```
a = 4*i;
b = c;
e = 4*i;
```

   pode ser otimizado para

```
a = 4*i;
b = c;
e = a;
```

   A subexpressão comum é `4*i`, sendo que o valor de `i` não é modificado entre as duas avaliações. Freqüentemente, o valor da subexpressão é colocado em uma variável temporária `t`:

```
t = 4*i
a = t;
b = c;
e = t;
```

   Esta otimização pode ser local a um bloco básico ou global, envolvendo vários blocos. Neste último caso, será necessário empregar algoritimos de análise de fluxo de execução para descobrir quais são as subexpressões comuns do programa. Veja o exemplo abaixo:

```
a = 4*i;
if ( i > 10 ) {
  i++;
  b = 4*i;
  }
else
  c = 4*i;
```

   Este código é transformado em

```
a = 4*i;
if ( i > 10 ) {
  i++;
  b = 4*i;
  }
else
  c = a;
```

2. Propagação de Cópia (*Copy Propagation*) .

   Sempre que houver uma atribuição

   ```
   b = c;
   ```

   a variável `c` poderá substituir `b` após esta instrução, desde que nenhuma das duas variáveis mude de valor.

   ```
   a = b;
   c = a + x;
   ```

   ```
   a = b;
   c = b + x;
   ```

   Com esta otimização, esta atribuição poderá tornar-se desnecessária e ser eliminada.

3. Eliminação de Código Morto (*Dead Code Elimination*)

   Código Morto é o código que nunca será executado, independente do fluxo de execução do programa.

   ```
   int f (int n )
   {
     int i = 0;

     while ( i < n ) {
       if ( g == h ) {
         break;
         g = 1;  // morto
         }
       i++;
       g--;
       }
     return g;
     g++;      // morto
     }
   ```

   Esta função pode ser otimizada para

   ```
   int f (int n )
   {
     int i = 0;

     while ( i < n ) {
       if ( g == h )
         break;
       i++;
       g--;
       }
   ```

```
  return g;
  }
```

Código morto pode ser identificado fazendo-se uma busca no grafo de fluxo de execução da função, começando-se na primeira instrução. Os blocos básicos não alcançados por esta busca nunca poderão ser executados. Como exemplo, traduziremos o programa acima para assembler:

```
     mov i, 0
     goto L1
L2:  cmp g, h
     goto<> L3
     goto L4
     mov g, 1
L3:  add i, 1
     sub g, 1
L1:  cmp i, n
     goto< L2
L4:  ret
     add g, 1
```

O grafo de fluxo de execução está na Figura 5.2. Observe que os blocos básicos `B4` e `B8` nunca podem ser executados se a execução começar em `B1`. Neste caso específico, nenhuma flecha chega a `B4` ou `B8`. Mas eles poderiam ser código morto mesmo se houvesse referências a eles. Examine o trecho de código a seguir:

```
return 1;
L1:  goto L2
     g = 1;
L2:  g++;
     goto L1;
```

4. Avaliação de Expressões Constantes.

```
const
  Max = 100*20,
  Tam = Max + 1;

a = 1 + a + 3;
```

```
const
  Max = 2000,
  Tam = 2001;

a = a + 4;
```

Figure 5.2: Grafo do Fluxo de Execução com código morto

Esta avaliação pode ser combinada com eliminação de código morto:

```
#define debug 0
...
if ( debug )
  g = 1;
else
  g = -1;
```

```
#define debug 0
...
g = -1;
```

5. Fatoração de Código

   Esta é uma otimização que troca velocidade por espaço, poupando este último. Duas seqüencias de instruções idênticas no código fonte causam a geração de apenas uma seqüencia de instruções no executável.

```
gets(s);
while ( *s != '\n' ) {
  cout << strupr(s) << endl;
  gets(s)  ;
  }
```

```
     goto L1;
L2:  cout << strupr(s) << endl;
L1:  gets(s);
     if ( *s != '\n' ) goto L2;
```

Em S2:

```
i = i + 1;
while i < 10 do
  begin
  j = j + 1;
  i = i + 1;
  end
```

```
     goto L1
```

```
L2:  add j, 1
L1:  add i, 1
     cmp i, 10
     goto < L2
```

Esta otimização é comum em `switch`'s:

```
switch (n) {
  case 1:
    f();
    g();
    puts(s);
    i++;
    break;
  case 2:
    write(fp);
    break;
  case 3:
    g();
    puts(s);
    i++;
  }
```

```
switch (n) {
  case 1:
    f();
  case 3:
    g();
    puts(s);
    i++;
    break;
  case 2:
    write(fp);
  }
```

6. Otimizações de `if`'s e `switch`'s

   Uma seqüencia de `if`'s aninhados como

```
if ( n == 1 )
  S1;
else if ( n == 2 )
```

```
  S2;
else if ( n == 3 )
  S3;
else
  S4;
```

onde n é um inteiro, pode ser otimizada para um comando `switch`:

```
switch (n) {
  case 1:
    S1;
    break;
  case 2:
    S2;
    break;
  case 3:
    S3;
    break;
  default:
    S4;
  }
```

A tradução do `switch` para assembler descrita anteriormente pode ser otimizada através do comando `goto v[i]` que permite desviar para um dos endereços armazenados em um vetor. O comando `switch` acima pode ser traduzido para:

```
     cmp n, 1
     goto<  L1
     cmp n, 3
     goto> L1
     goto ender[n]
L2:  codigo para S1
     goto fim
L3:  codigo para S2
     goto fim
L4:  codigo para S3
     goto fim
L1:  codigo para S4
fim:
```

Admitimos que o vetor `ender` tenha sido inicializado com os endereços L2, L3 e L4.

Nos casos em que existirem muitas opções do `switch` e os números presentes no `case` não estiverem em ordem, pode-se utilizar uma tabela *hash* para obter o endereço fornecendo-se o valor de `n` (ou a expressão do `switch`) como chave:

```
calcula a funcao hash usando n
goto R0
...
```

Na tradução acima, admitimos que em `R0` foi colocado o resultado do cálculo da função *hash*.

Existem algoritmos que geram uma função *hash* dado um conjunto de números ou *strings* como entrada. A tabela *hash* gerada a partir dos números poderá ser:

- perfeita, quando a função *hash* não colocar mais de um elemento em cada posição da tabela. Isto é, se `v` for a tabela, `v[i]` não apontará para um lista com mais de um nó;
- mínima, quando cada `v[i]` apontar para uma lista com pelo menos um nó.

Em uma tabela *hash* perfeita e mínima cada posição da tabela aponta para uma lista de exatamente um elemento. Neste caso, uma tabela *hash* para `k` elementos será implementada como um vetor de tamanho `k`.

7. Eliminação de Variáveis Inúteis

   Uma variável local a um procedimento será inútil se ela for usada apenas do lado esquerdo de atribuições ou não for utilizada de modo algum. Na função:

```
int fat( int n )
{
  int i, j, p, k;

  i = p = 1;
  for ( j = 2; j <= n; j++ )
    p *= j;
  return p;
  }
```

   as variáveis `i` e `k` são inúteis e podem ser eliminadas do código, resultando em

```
int fat( int n )
{
  int  j, p;

  p = 1;
  for ( j = 2; j <= n; j ++ )
    p *= j;
  return p;
  }
```

5.4 Otimizações de Laços

1. Movimentação de Código (*Code Motion*)

Expressões que são constante dentro de laços podem ser avaliadas antes do laço e o resultado reaproveitado. Exemplos:

```
i = 0;
while i < n - 1 do
  begin
  write(i);
  i = i + 1;
  end
```

```
i = 0;
t1 = n - 1;
while i < t1 do
  begin
  write(i);
  i = i + 1;
  end
```

O código

```
s = 0;
for ( i = 0; i < n; i++ )
  s += a*b/i;
```

pode ser otimizado para

```
s = 0;
t1 = a*b;
for ( i = 0; i < n; i++ )
  s += t1/i;
```

2. Redução em Poder (*Strengh Reduction*)

   Redução em poder refere-se a transformar operações lentas (como multiplicações) em operações rápidas (como somas) dentro de laços. A cada iteração é aproveitado o resultado obtido pela iteração anterior, eliminando a necessidade de operações mais complexas. A multiplicação `4*i` dentro do laço

   ```
   for ( i = 0; i < n; i++ )
     f( 4*i );
   ```

   pode ser transformada em soma:

```
t = 0;
for ( i = 0; i < n; i++ ) {
  f(t);
  t += 4;
  }
```

A variável `t` é chamada de variável de indução.

Existe uma operação de multiplicação implícita quando vetores são indexados. Redução em poder pode também ser utilizado neste caso.

```
floot v[ Max ];
for ( i = 0; i < Max; i++ )
  v[i] = 0;
```

```
floot v[ Max ], *p;

p = v;
for ( i = 0; i < Max; i++ )
  *p++ = 0;
```

Contudo, em algumas máquinas a primeira forma será mais rápida.

3. Supressão de Testes de Limites Redundantes

   Alguns compiladores inserem testes que conferem se a variável (ou expressão) que indexa um vetor está dentro dos limites permitidos. Se o vetor `v` for declarado como:

   ```
   var v : array(integer)[100];
   ```

   a atribuição

   ```
   v[i] = a;
   ```

   será traduzida para

   ```
   cmp i, 0
   goto< Erro
   cmp i , 100
   goto>= Erro
   mov v[i], a
   ...
   Erro:
   ```

   Na posição do *label* `Erro` estará um código que imprime uma mensagem de erro e termina o programa. Esta mensagem poderia informar o número da linha do código fonte onde aconteceu o erro, o valor de `i` e o nome do vetor.

   Uma alternativa a imprimir uma mensagem com erro é sinalizar uma exceção, se a linguagem suportar esta construção. Deste modo, a exceção poderia ser tratada pelo programa evitando o seu término forçado.

Observe que as instruções que se seguem ao *label* `Erro` que testam se `i >= 0` e `i < 100` fazem parte do sistema de tempo de execução, pois pertencem ao código gerado mas não foram inseridas diretamente pelo programador.

Se um vetor for indexado pela variável de repetição de um laço, os testes de limites poderão ser feitos uma única vez antes da execução do laço. Por exemplo,

```
  var  v : array(integer)[30];
  ...
begin
i = 0;
while i <= 12 do
  begin
  v[i] = 1;
  i = i + 1;
  end
...
end
```

Normalmente traduzido para

```
     mov i, 0
     goto L1
L2:  cmp i, 0
     goto< Erro
     cmp i, 30
     goto>= Erro
     mov v[i], 1
     add i, 1
L1:  cmp i, 12
     goto<= L2
     ...
     Erro:
```

pode ser otimizada para

```
     mov i, 0
     cmp 0, 0
     goto< Erro
     cmp 12, 30
     goto>= Erro
     goto L1
L2:  mov v[i], 1
     add i, 1
L1:  cmp i, 12
     goto<= L2
```

Como as comparações utilizam constantes, uma otimização a mais resulta em

```
     mov i, 0
     goto L1
```

```
L2:  mov v[i], 1
     add i, 1
L1:  cmp i, 12
     goto<= L2
```

Este último passo não seria possível ser `i` fosse inicializado com uma variável e o limite superior fosse também uma variável:

```
i = k1;
while i <= k2 do
  begin
  v[i] = 1;
  i = i + 1;
  end
```

4. Desdobramento de Laço (*Loop Unrolling*)

   Quando o número de repetições de um laço for conhecido em tempo de compilação, pode-se eliminar o laço e gerar o código de dentro da repetição o número de vezes em que o laço seria executado.

```
i = 0;
while i < 4 do
  begin
  s = s + v[i];
  i = i + 1;
  end
```

```
i = 0;
s = s + v[i];
i = i + 1;
s = s + v[i];
i = i + 1;
s = s + v[i];
i = i + 1;
s = s + v[i];
i = i + 1;
```

Um bom compilador poderia ainda otimizar este código para

```
i = 4;
s = s + v[0];
s = s + v[1];
s = s + v[2];
s = s + v[3];
```

e mesmo para

```
i = 4;
s = s + v[0] + v[1] + v[2] + v[3];
```

Se o número de vezes que o laço será executado não for conhecido em tempo de compilação, o corpo do laço pode ser duplicado, permitindo outras otimizações como eliminação de subexpressões comuns. Pittman e Peters [7] citam como exemplo o código

```
while ( a < b ) {
  b = b - a*k;
  a++;
  }
```

transformado em

```
while ( 1 ) {
  if ( a >= b )
    break;
  b = b - a*k;
  a++;
  if ( a >= b )
    break;
  b = b - a*k;
  a++;
  }
```

O valor do primeiro cálculo de `a*k` pode ser colocado em uma variável temporária `t1` e reutilizado na segunda atribuição

```
b = b - a*k;
```

que pode ser modificada para

```
b = b - t1 + k;
```

já que

```
a1*k = (a0 + 1)*k = a0*k + k = t1 + k
```

onde `a0` e `a1` correspondem aos dois valores da variável `a` neste laço.

Se o número de repetições for conhecido e par, pode-se duplicar o corpo do laço reduzindo-se pela metade o número de testes de fim de laço:

```
for ( i = 0; i < 100; i++ )
  s[i] = 0;
```

```
     i = 0;
     goto L1;
L2:  s[i] = 0;
     i++;
```

```
     s[i] = 0;
     i++;
L1:  if ( i < 100 ) goto L2
```

5.5 Otimizações com Variáveis

1. Colocação de Variáveis em Registradores

   A manipulação de registradores é muito mais rápida do que a manipulação de memória RAM. Por esta razão, é importante colocar as variáveis mais usadas de cada procedimento[2] em registradores. Em geral, as variáveis mais usadas são aquelas empregadas nos laços mais internos. No exemplo

   ```
   for ( i = 0; i < n; i++ )
     for ( j = 0; j < n; j++ )
       for ( k = 0; k < n; k++ )
         s[i][j][k] = 0;
   ```

   as variáveis mais utilizadas são, na ordem,

   ```
   k > j > i == s
   ```

   Assim, se houver apenas dois registradores disponíveis, eles serão alocados para `k` e `j`. O compilador pode otimizar este código por "Movimentação de Expressões Constantes". O endereço de `s[i][j]` é uma constante para o último laço, o do `k`:

   ```
   int *p;
   for ( i = 0; i < n; i++ )
     for ( j = 0; j < n; j++ ) {
       p = &s[i][j];
       for ( k = 0; k < n; k++ )
         *p++ = 0;
       }
   ```

   Neste caso, melhor seria colocar `p` e `k` em registradores.

   Registradores não possuem endereço e, portanto, uma variável que tem o seu endereço tomado (com & em C++) não pode ser colocada em um registrador.

   Suponha que existam dois procedimentos `P` e `Q` sendo que ambos utilizam os registradores `R0` e `R1` como variáveis locais. Após `P` chamar `Q`, os valores dos registradores `R0` e `R1` utilizados em `P` terão sido alterados por `Q`. Então, antes de chamar `Q`, o procedimento `P` deve empilhar estes registradores:

   ```
   push R0
   push R1
   call Q
   pop R1
   pop R0
   ```

[2] Chamaremos procedimentos qualquer subrotina, rotina ou função.

Ou `Q` deve salvar estes registradores antes de usá-los.

2. Reuso de Registradores e Variáveis Locais/Temporárias

   Um registrador/variável está *vivo* do ponto em que recebe um valor ao ponto onde é utilizado pela última vez. Se duas variáveis locais a uma subrotina nunca estão vivas ao mesmo tempo, elas podem ocupar a mesma posição de memória ou registrador. Assim, no código

```
void f()
{
  int i, j;

  for ( i = 0; i < 10; i++ )
    cout << i << endl;
  for ( j = 10; j > 0; j-- )
    cout << j << endl;

  }
```

   `i` e `j` podem ser a mesma variável:

```
void f()
{
  int i;

  for ( i = 0; i < 10; i++ )
    cout << i << endl;
  for ( i = 10; i > 0; i-- )
    cout << i << endl;
  }
```

   Esta técnica também é utilizada para diminuir o número de variáveis temporárias necessárias a uma subrotina.

5.6 Otimizações de Procedimentos

1. Passagem de Parâmetros/Valor de Retorno por Registradores

   O compilador pode adotar a convenção de passar os primeiros parâmetros de uma chamada de procedimento em determinados registradores utilizados apenas para esta finalidade. Sem esta otimização, os parâmetros são passados pela pilha, que é muito mais lenta. O mesmo raciocínio se aplica aos valores de retorno de funções.

2. Expansão em Linha de Procedimentos

   Procedimentos pequenos podem ser expandidos nos locais onde são chamados, eliminando a sobrecarga de uma chamada de subrotina. Por exemplo,

```
inline getValor()
{
  return valor;
```

```
  }
...
i = getValor() + 1;
```

é otimizado para

```
...
i = valor + 1;
```

Em C++, funções que devem ser substituídas em linha podem ser declaradas com a palavra chave `inline`, como mostrado acima.

Pode acontecer de haver chamadas recursivas entre as rotinas "inline". Neste caso, uma das rotinas não pode ser expandida.

Alguns compiladores expandem em linha funções de bibliotecas como `memset`, `strlen`, `strcpy`, etc. Mesmo que o usuário não declare alguma função como `inline` em C++, ela poderá ser expandida automaticamente pelo compilador.

Esta otimização é muito importante. Tipicamente, 40% do tempo de execução de um programa é gasto em chamadas de subrotinas. Cada chamada envolve: a) passagem de parâmetros, b) empilhamento do endereço de retorno, c) salto para a função, d) salvamento e inicialização de um registrador que permitirá manipular as variáveis locais e) alocação das variáveis locais f) destruição das variáveis locais g) salto para o endereço de retorno.

Entre e) e f) acontece a execução do corpo da função. Quando uma função for colocada em linha, os passos b)-g) e talvez a) serão eliminados. Esta otimização é particularmente importante porque parte considerável das funções é chamada apenas uma única vez em todo o programa. Assim, o uso intensivo desta otimização pode até tornar o programa menor.

3. Recursão de Cauda (*Tail Recursion*)

   Quando existir uma chamada de procedimento recursivo ao fim da execução do procedimento, esta poderá ser substituída por um desvio incondicional para o início do procedimento.

```
int E2 ()
{
  if ( lex->token == mais_smb ) {
    lex->nextToken();
    cout << "+";
    T();
    E2();
    }
  else
    if ( lex->token == menos_smb ) {
      lex->nextToken();
      cout << "-";
      T();
      E();
      }
  }
```

```
int E2()
{
  L:
  if ( lex->token == mais_smb ) {
    lex->nextToken();
    cout << "+";
    T();
    goto L;
    }
  else
    if ( lex->token == menos_smb ) {
      lex->nextToken();
      cout << "-";
      T();
      goto L;
      }
  }
```

Este exemplo é uma adaptação de um exemplo de Aho et al. [4]. Recursão de cauda poderá ser otimizada mesmo se o procedimento possuir parâmetros:

```
void P( int a )
{
  if ( a > 2 )
    P( a - 1 );
  else  if ( a == 2 )
    cout << "0" << endl;
  else
    P(10);
  }
```

```
void P( int a )
{
  L:
  if ( a > 2 ) {
    a = a - 1;
    goto L;
    }
  else
    if ( a == 2 )
      cout << "0" << end;
```

```
    else {
      a = 10;
      goto L;
      }
  }
```

Observe que

```
void P( int a )
{
  if ( a > 2 )
    P( a - 1 );
  else
    cout << "0" << endl;
  cout << "P" << endl;
  }
```

não pode ser otimizado porque há uma instrução após "`P( a - 1 )`".

4. Transformação de Variáveis Locais em Estáticas

   Quando um procedimento é chamado, deve-se alocar memória na pilha para as suas variáveis locais, que são manipuladas por meio do registrador `bp`. O código do início de um procedimento `P`, com duas variáveis locais, deve ser

   ```
   push bp
   mov bp, t
   add t, 2
   ```

   sendo que `t` é o registrador contendo o topo da pilha da máquina. A primeira variável local será manipulada por `bp[1]` e a segunda, por `bp[2]`. O valor de `bp` é salvo porque ele é utilizado também pelo procedimento que chamou `P`. Este valor é restaurado ao final da execução de `P`, juntamente com a desalocação das variáveis locais:

   ```
   sub t, 2
   pop bp
   ret
   ```

   A alocação e desalocação de variáveis locais na pilha é lenta e pode ser substituída, em alguns casos, por alocação estática, feita uma única vez antes do início da execução do programa.

   Se o procedimento não for direta ou indiretamente recursivo, haverá, no máximo, um conjunto de suas variáveis locais na pilha do computador. Sendo assim, as variáveis podem ser alocadas estaticamente. Se o procedimento for recursivo, não saberemos, em tempo de compilação, quantas vezes ele chamará direta ou indiretamente a si mesmo e, portanto, não sabemos quantos conjuntos de suas variáveis locais serão necessários em tempo de execução.

   Pode-se descobrir quais são os procedimentos recursivos de um programa através de uma busca em um grafo dirigido onde os vértices são os procedimentos e existe uma aresta de `v` para `w` se o procedimento `v` pode chamar `w` em tempo de execução. Em outras palavras, haverá aresta (`v`, `w`) se houver uma chamada

   ```
   w();
   ```

   em algum lugar do procedimento `v`.

Figure 5.3: Grafo de chamadas de procedimentos

Um exemplo de um grafo construído segundo esta regra é mostrado na Figura 5.3. Existe uma chamada de `m` a si mesmo e, portanto, `m` é recursivo. O procedimento `g` chama `p` que chama `q` que chama `g`, existindo então uma recursão `g-p-q-g`.

Nas frases acima, utilizamos "`x` chama `y`" para significar "no código fonte de `x` existe uma chamada a `y`". Talvez esta chamada nunca ocorra em tempo de execução, quaisquer que sejam os dados fornecidos ao programa. Como é impossível afirmar com certeza se `x` irá mesmo chamar `y`, admitiremos que isto *pode* acontecer.

Sempre que houver um círculo no grafo de chamadas, poderá haver recursão em tempo de execução e assim os procedimentos envolvidos em recursão devem ter alocação dinâmica de variáveis locais, na pilha. Os outros procedimentos podem utilizar alocação estática.

Observe que os procedimentos `f` e `r` nunca estarão na pilha ao mesmo tempo. Portanto, podemos alocar uma única área de memória para as variáveis de `f` e `r`. Este raciocínio pode ser estendido para todo o grafo.

Para explicar este ponto, considere um grafo representando um programa em C. A "raiz" do grafo é a função `main` e as folhas são procedimentos que não chamam ninguém. Se todas as funções forem recursivas, todas as variáveis locais devem ser alocadas dinamicamente na pilha.

Caso contrário, existe pelo menos uma função não recursiva. Considere que $\mathtt{p}_1$, $\mathtt{p}_2$, ..., $\mathtt{p}_n$ sejam todos os caminhos que começam em `main` e terminam em a) uma folha ou b) uma função recursiva. Sendo $\mathtt{esp(q}_j\mathtt{)}$ o número de bytes necessários para as variáveis locais de $\mathtt{q}_j$, o número $\mathtt{esp(p}_i\mathtt{)}$ de bytes necessários para as variáveis locais de todas as funções no caminho $\mathtt{p}_i = \mathtt{q}_1\mathtt{q}_2\mathtt{...q}_k$ é dado por

$$\mathtt{esp(p}_i\mathtt{)} = \sum_{j=1}^{k} \mathtt{esp(q}_j\mathtt{)}$$

É necessário alocar estaticamente para todo o programa um número de bytes igual a

$$\mathtt{max(esp(p}i\mathtt{))},\ 1 \leq i \leq n$$

Como exemplo, o grafo de chamadas da Figura 5.4 mostra ao lado de cada procedimento quantos bytes são necessários para as variáveis locais. Para este programa, o caminho que necessitará de mais memória será

$$\mathtt{main} \Longrightarrow \mathtt{g} \Longrightarrow \mathtt{n}$$

em um total de 38 bytes.

Figure 5.4: Grafo de chamadas de procedimentos

5.7 Dificuldades com Otimização de Código

Algoritmos de otimização de código são complexos e as transformações que eles fazem podem não corresponder precisamente à definição semântica da linguagem utilizada. Isto é, uma otimização pode transformar um código em outro mais eficiente mas que não é equivalente ao primeiro.

Veremos a seguir algumas transformações incorretas que um otimizador de código pode fazer.

i) Ao avaliar uma expressão constante, como em

```
x = 3.5/7*9 + 3/2;
```

o compilador ou otimizador deve empregar as mesmas regras de avaliação que as definidas pela linguagem. A maneira mais segura de avaliar expressões é gerar um pequeno programa que avalie esta expressão. Assim, se a linguagem utilizada for C, pode-se gerar o programa

```
#include <stdio.h>

void main()
{
  printf("%f\n", 3.5/7*9 + 3/2 );
  }
```

executá-lo, tomar o resultado e substituir a atribuição à variável x pela atribuição ao resultado.

ii) Uma multiplicação `2*n` pode ser transformada em `n << 1`. Contudo, haverá um erro se a instrução "rolamento à esquerda" (que é `<<`) da máquina alvo colocar o último bit do número na primeira posicão. Por exemplo, se `n` for

```
10011010
```

o resultado de `n << 1` poderá ser

```
00110101
```

iii) A movimentação de expressões constantes para fora de laços pode retirar testes de proteção para a expressão contra divisão por zero, estouro de limites, etc. Como exemplo,

```
s = 0;
```

```
for ( i = 0; i < n; i++ )
  if ( b != 0 )
    s += v[i] + a/b;
```

poderia ser incorretamente otimizado para

```
s = 0;
t1 = a/b;
for ( i = 0; i < n; i++ )
  if ( b != 0 )
    s += v[i] + t1;
```

iv) Aliás (*alias*) é o uso de dois nomes para uma mesma posição de memória. Como exemplo, no código

```
void  m( int &a, int &b )
{

  a = 2;
  b = 5;
  b = a*2;
  }
```

haverá um aliás se `m` for chamado por

```
m( x, x );
```

Os parâmetros `a` e `b` irão referenciar `x`. Por este motivo, a função `m` não pode ser otimizada para

```
void m ( int &a, int &b )
{
  a = 2;
  b = 4;
  }
```

Aliás pode acontecer em C++ na presença de variáveis passadas por referência e ponteiros. Então, sempre que houver variáveis deste tipo temos que assumir que elas podem ser modificadas por qualquer atribuição ou chamada de função, o que impede muitas otimizações: variáveis que eram constantes em determinado trecho (como `a` no exemplo acima) não podem mais ser consideradas como tal.

A linguagem C++ permite conversão de ponteiros de um tipo para outro desde que se utilize um *cast*:

```
  char *s;
  float f;
  int *p;
...
p = (int *) s;
...
p = &f;
```

Isto significa que um ponteiro pode referenciar variáveis e áreas de memória de qualquer tipo. Deste modo, uma atribuição

```
*p = 0;
```

pode, potencialmente, alterar qualquer variável, vetor ou objeto, dificultando a realização de otimizações como eliminação de subexpressões comuns, propagação de variáveis e movimentação de expressões constantes para fora de laços.

No exemplo

```
void m( int *p, int i )
{
  int a = i, b, c;

  if ( i > 0 ) p = &a;
  c = 3*a;
  *p = 5;
  b = 3*a;
  ...
  }
```

A subexpressão `3*a` não pode ser calculada uma única vez porque o valor de `a` pode ser alterado entre as atribuições "`c = 3*a`" e "`b = 3*a`". Se o endereço de uma variável for tomado, como em "`p = &a`", deve-se admitir que o seu conteúdo pode ser modificado indiretamente. Observe que o uso de ponteiros não impede sempre o compilador de fazer otimizações, mas torna a análise do que é constante em um certo trecho de código muito difícil.

Se a função `m` fosse definida como

```
void m( int *p, int i )
{
  int a = i, b, c;

  c = 3*a;
  b = 3*a;
  if ( i > 0 ) p = &a;
  *p = 5;
  ...
  }
```

a expressão `3*a` poderia ser calculada uma única vez porque as atribuições a `b` e `c` precedem, no grafo de fluxo de execução desta função, à tomada de endereço de `a`.

Se houver mais de um vetor como parâmetro, como em

```
void q( int v[], int w[], int n )
{
  int i = 0;

  v[0] = 3*w[0];
  a = 3*w[0];
  ...
  }
```

o compilador deve considerar que a escrita em uma posição de `v` pode alterar outra posição de `w`, pois os dois vetores podem se referir à mesma área de memória ou áreas sobrepostas. Assim, a função acima não pode ser otimizada para

```
void q( int v[], int w[], int n )
{
  int i = 0, t1;

  v[0] = t1 = 3*w[0];
  a = t1;
  ...
  }
```

onde `t1` é uma variável temporária, porque esta função poderia ser chamada como

```
int s[100];
...
q( s, s, 100 );
```

e, neste caso, `v[0]` seria igual a `w[0]`.

Em muitos compiladores, uma opção de compilação "Assume no alias" pode ser ligada quando o programador tiver certeza de que não haverá nenhum aliás em chamadas de função. Neste caso, a função `q` poderia ser otimizada porque o programador estaria afirmando que chamadas como "`q( s, s, 100)`" nunca ocorrerão. Claramente, esta opção é muito perigosa e deve ser ligada em muito poucos casos.

Appendix A

A Linguagem S2

A linguagem S2 é um pequeno subconjunto de Pascal com algumas modificações. O nome S2 provém de SS, Super Simples. A linguagem Simples é um superconjunto de S2 com suporte a orientação a objetos. Ela será estudada na parte 2 desta apostila. Simples é também um subconjunto de Green [8].

Um exemplo de um pequeno programa em S2 que imprime os `n` primeiros números inteiros, `n` lido do teclado, é mostrado na Figura A.1. Este programa possui os principais elementos da linguagem. O programa começa com a declaração das variáveis globais, parte que é opcional. Em seguida, há o corpo do programa entre `begin` e `end`, com zero ou mais instruções. Os comandos são semelhantes aos de Pascal, exceto que o `if` é terminado por `endif` e não há "." após o `end` que termina o programa. O ";" termina as atribuições e comandos `read` e `write`. Como qualquer outra linguagem de programação, os terminais do programa são separados por branco, fim de linha ou caráter de tabulação.

A seguir detalhamos o significado de cada um dos elementos de S2.

A.1 Comentários

Comentários na linguagem são colocados entre "{" e "}". Comentários aninhados não são permitidos, como

```
{  comentario { outro comentario }  fim primeiro }
```

A linguagem S2 também admite comentários do tipo `//` iguais aos de C++. Qualquer coisa após `//` até o fim da linha é ignorado pelo compilador. Os símbolos { e } dentro de um comentário iniciado por `//` não significam comentário. O mesmo se aplica a `//` entre { e }.

A.2 Tipos e Literais Básicos

Existem apenas dois tipos em S2, `integer` e `boolean`. Literais do tipo `integer` devem estar entre os limites `0` e `32767`, sendo que qualquer número de zeros antes de um número é permitido. Assim, os números

```
00000000000001
00000000000000
```

são válidos. O tipo `boolean` possui apenas dois valores: `false` e `true`.

Os operadores `<`, `<=`, `>`, `>=`, `==`, `<>` de comparação podem ser aplicados a valores inteiros ou booleanos sendo que `false < true`. Naturalmente, ambos os operados de uma destas operações devem ser do mesmo tipo.

```
  var i, n : integer;
begin
read(n);
if n > 0
then
  i = 1;
  while i <= n do
    begin
    write(i);
    i = i + 1;
    end
endif
end
```

Figure A.1: Um programa em S2

`and`	`begin`	`boolean`
`do`	`else`	`end`
`endif`	`false`	`if`
`integer`	`not`	`or`
`read`	`then`	`true`
`var`	`while`	`write`

Figure A.2: As palavras chave de S2

Os operadores `+`, `*`, `-` e `/` aplicam-se a valores inteiros resultando em inteiros. A semântica destes operadores é a mesma dos operadores da linguagem C.

Os operadores binários `and` e `or` e o unário `not` aceitam operandos booleanos e possuem o significado usual. A avaliação da expressão

```
expr1 and expr2
```

começa em `expr1`. Se `expr1` for `false`, toda a expressão será considerada falsa, mesmo sem o cálculo de `expr2`. Se for `true`, `expr2` será avaliada.

A expressão

```
expr1 or expr2
```

será considerada `true` se `expr1` for `true`. Caso contrário, esta expressão terá o valor de `expr2`.

A.3 Identificadores

Identificadores são formados por letras, dígitos e o caráter sublinhado ("_"), iniciando-se por uma letra. Exemplos de identificadores válidos são:

```
getNum   x0   y1   get_Num
```

Os identificadores

```
_main    3ab    get$Num     write
```

são ilegais. "`write`" é ilegal por ser uma palavra chave. A lista das palavras chave da linguagem é exibida na Figura A.2.

Os primeiros 31 caracteres de um identificador são significativos e letras maiúsculas e minúsculas são consideradas diferentes. Assim, os identificadores

```
a12345678901234567890123456789012345678901Um
a12345678901234567890123456789012345678901Dois
```

são iguais. Todos os identificadores devem ser declarados antes de serem usados e nenhum pode ser declarado duas vezes.

A.4 Atribuição

A atribuição de uma expressão `expr` a uma variável `aa` é

```
aa = expr
```

onde o tipo de `aa` e `expr` devem ser iguais.

A.5 Comandos de Decisão

O comando `if` de S2 possui a forma

```
if expr
then
  StatementList
else
  StatementList
endif
```

onde a parte

```
else
  StatementList
endif
```

é opcional. `expr` é uma expressão booleana e `StatementList` é uma lista de zero ou mais comandos.

A.6 Comandos de Repetição

O comando

```
while expr do
  Statement;
```

repete `Statement` enquanto a avaliação da expressão booleana `expr` resultar em `true`. Este comando também possui a forma

```
while expr do
  begin
  StatementList
  end
```

onde `StatementList` possui zero ou mais comandos.

A.7 Entrada e Saída

A entrada de dados é feita pelo comando `read`:

```
read( IdList );
```

onde `IdList` é uma lista de uma ou mais variáveis inteiras. O comando

`read( a`$_1$`, a`$_2$`, ... a`$_n$ `)`
é equivalente a
`read( a`$_1$ `)`
`read( a`$_2$ `)`
`...`
`read( a`$_n$ `)`
onde `read( a )` é equivalente ao código
```
gets(s);
sscanf(s, "%d", &a);
```
em C. Isto é, cada variável é lida em uma linha separada da entrada padrão.

O comando
`write( expr`$_1$`, expr`$_2$`, ... expr`$_n$ `)`
escreve as expressões na saída padrão, sendo equivalente a
`write( expr`$_1$ `)`
`write( expr`$_2$ `)`
`...`
`write( expr`$_n$ `)`
O número `n` de expressões deve ser maior do que zero. O comando `write(expr)` é equivalente ao código
```
printf("%d ", expr);
```
em C. Apenas expressões inteiras podem ser parâmetros de `write`.

A.8 A Gramática de S2

Esta seção define a gramática da linguagem. As palavras reservadas e símbolos da linguagem são mostrados entre “ e ”. Qualquer seqüência de símbolos entre { e } pode ser repetida zero ou mais vezes e qualquer seqüência de símbolos entre [e] é opcional. O prefixo Un em um nome significa a união de duas ou mais regras.

Assignment	::= Id “=” Expression
BasicType	::= “integer” | “boolean”
BasicValue	::= IntValue | BooleanValue
Block	::= “begin” StatementList “end”
BooleanValue	::= “true” | “false”
Digit	::= “0” | ... | “9”
Expression	::= SimpleExpression [Relation SimpleExpression]
ExpressionList	::= Expression { “,” Expression }
Factor	::= BasicValue | Id | “(” Expression “)” | “not” Factor
HighOperator	::= “∗” | “/” | “and”
Id	::= Letter { Letter | Digit | “_” }
IdList	::= Id { “,” Id }
IfStat	::= “if” Expression “then” StatementList [“else” StatementList] “endif”
IntValue	::= Digit { Digit }
Letter	::= “A” | ... | “Z” | “a” | ... | “z”
LocalDec	::= “var” VarDec { VarDec }
LowOperator	::= “+” | “−” | “or”

Program	::= [LocalDec] Block
ReadStat	::= “read” “(” IdList “)”
Relation	::= “==” | “<” | “>” | “<=” | “>=” | “<>”
Signal	::= “+” | “−”
SimpleExpression	::= [Signal] Term { LowOperator Term }
Statement	::= Assignment “;” | IfStat | WhileStat | ReadStat “;” | WriteStat “;” | “;”
StatementList	::= { Statement }
Term	::= Factor { HighOperator Factor }
UnStatBlock	::= Statement | Block
WriteStat	::= “write” “(” ExpressionList “)”
VarDec	::= IdList “:” BasicType “;”
WhileStat	::= “while” Expression “do” UnStatBlock

Bibliography

[1] Stroustrup, Bjarne. *The C++ Programming Language*. Second Edition, Addison-Wesley, 1991.

[2] Lippman, Stanley B. *C++ Primer*. Addison-Wesley, 1991.

[3] Deitel, H.M. e Deitel P.J. *C++ How to Program*. Prentice-Hall, 1994.

[4] Aho, Alfred e Sethi, Rave e Ullman, Jeffrey. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

[5] Gamma, Erich; Helm, Richard; Johnson, Ralph e Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994.

[6] Zorzo, Sérgio Donizetti. Notas de aula de Construção de Compiladores, 1995.

[7] Pittman, Thomas; Peter, James. *The Art of Compiler Design: Theory and Practice*. Prentice Hall, 1992.

[8] Guimarães, José de Oliveira. *The Green Language*. Disponível em http://www.dc.ufscar.br/ jose/green/green.htm.