Compiladores

Otimização de Código

José Lucas Rangel
21 nov 00

Introdução

Várias técnicas e várias tarefas se reúnem sob o nome de Otimização. Para começar, o próprio nome é um engano: pode-se demonstrar que, para qualquer critério de qualidade razoável, é impossível construir um programa "otimizador", isto é, um programa que recebe como entrada um programa P e constrói um programa P' equivalente que é o melhor possível, segundo o critério considerado. O que se pode construir são programas que melhoram outros programas, de maneira que o programa P' é, na maioria das vezes, melhor, segundo o critério especificado do que o programa P original. A razão para essa impossibilidade é a mesma que se encontra quando se estuda, por exemplo em LFA (Linguagens Formais e Autômatos), o "problema da parada": um programa (procedimento, máquina de Turing, etc.) não pode obter informação suficiente sobre todas as possíveis formas de execução de outro programa (procedimento, máquina de Turing, etc.).

Normalmente, é muito fácil fazer uma otimização em um programa, ou seja, uma transformação que o transforma em outro melhor. O difícil é sempre obter a informação necessária que garante que a otimização pode realmente ser aplicada, sem modificar em nenhum caso o funcionamento do programa.

Por exemplo, considere um trecho de programa em que aparece um comando `x=a+b;` Este programa pode ser melhorado (torna-se mais rápido e menor) se este comando for retirado. Mas para que o funcionamento do programa não seja alterado, é necessário mostrar uma entre diversas propriedades, algumas das quais aparecem a seguir:

1. o comando `x=a+b;` nunca é executado. Por exemplo, está em seguida a um `if` cuja condição nunca é satisfeita:
   ```
   if(0)
      x=a+b;
   ```
2. o comando é inútil, porque todas as vezes que é executado, `x` recebe exatamente o mesmo valor que tinha antes . Por exemplo, podemos retirar o segundo comando na seqüência
   ```
   x=a+b;
   x=a+b;
   ```
3. o comando é inútil, porque nenhum dos comandos executados posteriormente, usa o valor da variável `x`. Por exemplo, se tivermos no programa uma função
   ```
   int f(int z) {
     int x;
     ...
     x=a+b;
   }
   ```

o valor de x nunca será utilizado, simplesmente porque após a saída da função, a variável `x`, que é local à função, não mais existe.

Este exemplo mostra que não seria prático tentar otimizar um programa tentando sucessivamente eliminar todos os seus comandos, um de cada vez. Note que as condições para cada eliminação dependem do comando, de sua posição, e, de certa maneira, de todos os comandos restantes do programa. Para construir um otimizador de utilidade na prática, precisamos identificar oportunidades para otimização que sejam produtivas em situações correntes. Por exemplo, nenhum programador escreveria *intencionalmente* nenhum dos três trechos de código usados como exemplo acima, de forma que não valeria a pena procurar exatamente estas situações.

Outro exemplo de otimização que é citado freqüentemente é a retirada de comandos de um comando de repetição (um *loop*). Por exemplo, se encontrarmos em um loop um comando cujo efeito é independente do loop, pode valer a pena retirá-lo do loop, para que ele seja executado apenas uma vez, em vez das muitas vezes que se presume que será executado o código de dentro do loop. Por exemplo, se `N` tem o valor 100,

```
for (i=0; i<N; i++) {
   a=j+5;
   f(a*i);
}
```

poderia ser melhorado retirando-se o comando `a=j+5;` do `for`. Ficaríamos com

```
a=j+5;
for (i=0; i<N; i++)
   f(a*i);
```

e 100 somas a menos seriam executadas. Por outro lado, se `N=0`, o programa foi “piorado”, ou “pessimizado”, porque o comando `a=j+5;` que era executado 0 vezes, passou a ser executado 1 vez. Mas pode haver um problema maior: se a variável `a` é usada após o loop, em vez de seu valor original, seu valor será, incorretamente, o resultado dado pela atribuição.

Depende muito da finalidade de um compilador o conjunto de otimizações que ele deve oferecer. Um compilador usado em um curso introdutório de programação não precisa de otimização, porque os programas são executados quase sempre apenas uma vez. (Em vez de otimização, este compilador deveria dar boas mensagens de erro, que pudessem auxiliar os principiantes na linguagem.) Por outro lado, um programa que vai ser compilado uma vez e executado muitas vezes deve ser otimizado tanto quanto possível. Neste caso estão programas de simulação, de previsão do tempo, e a maioria das aplicações numéricas.

Um outro problema é a quantidade de informação que se deseja manipular. Podemos examinar otimizações locais (em trechos pequenos de programas, por exemplo trechos sem desvios, ou seja, trechos em linha reta), otimizações em um nível intermediário (as otimizações são consideradas apenas em funções, módulos, ou classes, dependendo da linguagem) e otimizações globais (que consideram as inter-relações de todas as partes de um programa). A maioria dos compiladores oferece algumas otimizações do primeiro tipo, possivelmente combinadas com a fase de geração de código, e quando muito algumas otimizações de nível intermediário.

A maneira de tratar a otimização pode ser extremamente pragmática. Em um programa grande, a execução gasta 90% em 10% do código, e 10% do tempo nos 90% do código restantes. (Escolha suas percentagens, a literatura menciona valores de 90%-10% até 70%-30%.) Isto acontece porque em geral um programa é constituído de inicializações e

finalizações, entre as quais se encontra um conjunto de vários loops aninhados. O tempo maior de execução ("os 90%") é gasto no loop mais interno ("os 10%"). Existem ferramentas que registram o perfil de execução do programa (*profilers*), permitindo a identificação dos trechos mais executados, e a otimização pode se concentrar nestes trechos, ignorando para fins de otimização o restante do programa. Por essa razão muitos compiladores oferecem opções de compilação que podem ser ligadas e desligadas no código fonte, indicando para o compilador os trechos que o programador considera importantes o suficiente para serem otimizados. Por estas razões, muito do trabalho no desenvolvimento de técnicas de otimização é dedicado aos loops.

Neste texto introdutório, não vamos dar nenhuma definição formal de loop, mas vale a pena definir o que entendemos por *bloco básico*, ou "trecho de código em linha reta". Para dividir um programa (ou um trecho de programa) em representação intermediária em blocos básicos, temos o seguinte:

- primeiro comando inicia um bloco básico.
- Qualquer comando rotulado, ou que de alguma forma seja alvo de um comando de desvio inicia um novo bloco básico
- Qualquer comando de desvio, condicional ou incondicional termina um bloco básico.

ou seja, blocos básicos são trechos de programa maximais cujas instruções são sempre executadas em ordem (em linha reta), da primeira até a última. Maximal aqui quer dizer que é impossível acrescentar outra instrução mantendo a mesma propriedade.

Várias técnicas de otimização se aplicam a blocos básicos.

A otimização pode ser feita tipicamente em três ocasiões:

(1) na representação intermediária, antes da geração de código;
(2) durante o processo de geração de código, que já é gerado de forma otimizada;
(3) após a geração de código diretamente no código objeto.

As formas de otimização durante a geração de código estão discutidas no Capítulo de "Geração de Código". A maioria das otimizações que discutiremos aqui se aplica melhor na representação intermediária. A forma mais comum de otimização que se aplica direto ao código objeto é a otimização "peephole", que discutimos no capítulo de "Tradução Dirigida pela Sintaxe".

Uma observação final. Embora a otimização possa melhorar as características de um programa, não faz milagres, de forma que não se pode justificar um programa mal escrito, porque "vai ser otimizado". Em particular, normalmente a otimização não altera a ordem dos algoritmos, alterando apenas as constantes multiplicativas. Se existe para um problema um algoritmo O(n log n), não espere que um programa em que você usou um algoritmo $O(n^2)$ se transforme durante a otimização no algoritmo O(n log n).

Oportunidades de otimização.

Listamos aqui uma série de oportunidades para otimização que são aproveitadas em compiladores. Esta lista não é exaustiva, e visa apenas mostrar as possibilidades existentes.

Eliminação de sub-expressões comuns. Suponha que a mesma expressão (possivelmente uma sub-expressão de outra expressão maior) ocorre mais de uma vez em um trecho de

programa. Se as variáveis que ocorrem na expressão não tem seus valores alterados entre as duas ocorrências, é possível calcular seu valor apenas uma vez.

Exemplo:

```
...
x=a+b;
...
y=a+b;
...
```

Se os valores de `a` e de `b` não são alterados, é possível guardar o valor da expressão `a+b` em uma temporária (digamos `t1`) e usá-lo outra vez posteriormente.

```
...
t1=a+b;
x=t1;
...
y=t1;
...
```

ou, se a variável `x` ainda está disponível com o mesmo valor da segunda vez que a expressão é calculada,

```
...
x=a+b;
...
y=x;
...
```

dispensando o uso da temporária `t1`.

Note que, para garantir que os valores de `a`, `b` (e, se for o caso, `x`) não se alteram, é preciso examinar todos os comandos que podem ocorrer entre as duas avaliações da expressão. Se estes dois comandos não estão no mesmo bloco básico, isto significar que devemos verificar durante a execução do programa quais outros blocos básicos podem ter seus comandos executados, para garantir que os valores que nos interessam não são alterados.

Mesmo no caso em que ambos os comandos estão no mesmo bloco básico podemos ter problemas. Por exemplo, se uma das variáveis consideradas é uma referência (uma componente) de um array, digamos, `a[i]`, qualquer alteração de valor de uma componente `a[j]` do mesmo array entre as duas avaliações da expressão faz com que não mais possamos garantir que o valor de `a[i]` não se alterou. Isto acontece porque não podemos garantir, a não ser em casos muito particulares, que `i≠j`. O mesmo acontece com atribuição de valores a variáveis apontadas e com chamadas de funções, com efeitos colaterais desconhecidos. No caso de variáveis apontadas, devemos considerar, por segurança, que todas as variáveis do tipo apontado foram alteradas. Considere o exemplo

```
int *p, a, b;
...
x=a+b;
...
*p=...;
...
y=a+b;
...
```

Neste caso, não sabemos para qual inteiro `p` aponta, e portanto não podemos garantir que `*p` não é um pseudônimo para `a` ou para `b`.

No caso de uma chamada de função, em princípio podemos fazer uma otimização global, e examinar o código da função para determinar que efeitos colaterais ela pode ter. Isso é possível (e trabalhoso) quando o trecho que está sendo considerado para otimização e a função são compilados ao mesmo tempo. Mas, se acontecer que a função em questão foi compilada separadamente, fica ainda mais difícil determinar se uma chamada da função pode alterar os valores das variáveis que nos interessam.

Vamos mostrar posteriormente um algoritmo para o caso mais simples: código em linha reta, sem comandos "suspeitos" entre as ocorrências da expressão.

Eliminação de código morto. Supondo que um programa contenha código morto (*dead code*), isto é, código que não pode ser alcançado durante a execução de um programa, este programa pode ser otimizado pela remoção deste código. O código morto ser identificado em uma série de situações:

- ocorre após uma instrução de encerramento de um programa ou de uma função. Por exemplo após um comando `return;`

```
int f(int x) {
   return x++;        /* o incremento não será executado */
}
```

- ocorre após um teste com uma condição impossível de ser satisfeita. Por exemplo

```
#define DEBUG 0
...
if(DEBUG) {
   ...                /* este código não será executado */
}
```

- ocorre após um comando de desvio, e não é alvo de nenhum outro desvio.

```
   goto x;
   i=3;               /* este código não será executado */
   ...
x: ...
```

Normalmente, não se espera que um programador escreva código morto. Entretanto, isso pode acontecer em várias situações, uma das quais é um estágio intermediário da otimização de um programa, que sofreu algumas transformações que causaram a existência de código não alcançável.

O exemplo de código inserido para depuração que aparece acima, e que pode ser desligado ou ativado dependendo do valor na macro `#define` não é um bom exemplo, no caso de linguagens que oferecem uma opção de compilação condicional, que evita este

problema. No caso, a presença de código morto seria indicada exatamente pelos comandos de compilação condicional.

Renomeação de variáveis temporárias. Já vimos em várias ocasiões que as variáveis temporárias introduzidas durante a geração de código intermediário podem não ser estritamente necessárias. Normalmente esta eliminação é feita dando outros nomes para as variáveis (temporárias ou não) que vao guardar os valores temporários. Por exemplo, se tivermos no código fonte `x=a+b;`, o código intermediário será

```
t1=a+b;
x=t1;
```

e a variável `t1` pode ser eliminada.

Na eliminação de sub-expressões comuns, podemos ficar com várias variáveis desnecessárias, associadas às várias cópias da sub-expressão. Por exemplo, se tivermos no código fonte

```
x=a+b;
x=t1;
y=(a+b)*c;
z=d+(a+b);
```

termos no código intermediário

```
t1=a+b;
x=t1;
t2=a+b;
t3=t2*c;
y=t3;
t4=a+b;
t5=d+t4;
z=t5;
```

e, eliminando as duas últimas cópias de `a+b`,

```
t1=a+b;
x=t1;
t2=t1;
t3=t2*c;
y=t3;
t4=t1;
t5=d+t4;
z=t5;
```

de maneira que (neste exemplo) todas as variáveis temporárias podem ser eliminadas:

```
x=a+b;
y=x*c;
z=d+x;
```

Como as variáveis temporárias são (como o nome diz) temporárias, pode acontecer que uma variável temporária possa ser re-usada após uma ocorrência anterior. Isto é feito mudando os nomes de algumas ocorrências dessas variáveis. Por exemplo, se tivermos

```
x=(a+b)*(c+d);
y=(e*f)+(g*h);
```

o código intermediário seria

```
t1=a+b;
t2=c+d;
t3=t1*t2;
x=t3;
t4=e*f;
t5=g*h;
t6=t4+t5;
y=t5;
```

e poderíamos *renomear* `t4` como `t1`, `t5` como `t2` e eliminar `t3` e `t6`:

```
t1=a+b;
t2=c+d;
x=t1*t2;
t1=e*f;
t2=g*h;
y=t1+t2;
```

Uma solução menos óbvia seria

```
x=a+b;
t2=c+d;
x=x*t2;
y=e*f;
t2=g*h;
y=y+t2;
```

Esta solução usa `x` como variável temporária, porque o valor de `x` inicial é irrelevante. O mesmo acontece com `y`.

Transformações algébricas. Podemos aplicar algumas transformações baseadas em propriedades algébricas, como comutatividade, associatividade, identidade, etc. Por exemplo, como a soma é comutativa, podemos transformar `x=a+b*c;` em `x=b*c+a;` o que corresponde a trocar código como

```
Load b
Mult c
Store t1
Load a
Add t1
Store x
```

por

```
Load b
Mult c
Add a
Store x
```

que dispensa a temporária `t1`, e as instruções que a manipulam.

Transformações semelhantes podem ser baseadas na associatividade, permitindo trocar `x=(a+b)+(c+d);` por `x=(((a+b)+c)+d);` o que corresponde a trocar

```
Load a
Add b
Store t1
Load c
Add d
Store t2
Load t1
Add t2
Store x
```
por
```
Load a
Add b
Add c
Add d
Store x
```
dispensando o uso das temporárias `t1` e `t2`.

Entretanto, estas transformações só devem ser utilizadas com autorização explícita do usuário, uma vez que algumas destas transformações podem criar problemas para a convergência ou a estabilidade em relação a erros de arredondamento dos programas. Normalmente, quando a ordem de precedência é indicada explicitamente com o uso de parênteses, a ordem de avaliação dada não deve ser alterada. Este seria o caso de `x=(a+b)+(c+d);`, em que os parênteses são desnecessários.

Dobramento de constantes. Expressões ou sub-expressões compostas de valores constantes podem ser avaliadas em tempo de compilação (*dobradas*), evitando sua avaliação repetida em tempo de execução. Por exemplo, se tivermos

```
#define N 100
...
while (i<N-1) {
   ...
}
```
não há necessidade de calcular repetidamente o valor de `N-1`. Este valor pode ser pré-calculado, e substituído por `99`.

Em alguns casos, problemas de portabilidade podem exigir que a expressão constante seja avaliada na máquina em que a execução vai se realizar, porque a máquina em que a compilação se realiza não é a máquina alvo. Neste caso em vez de avaliar a expressão em tempo de compilação, ela é avaliada uma vez, logo no início da execução.

Redução de força. Há vários exemplos em que operações mais caras podem ser substituídas por operações mais baratas. Por exemplo, para calcular o comprimento da concatenação de duas cadeias, podemos somar os comprimentos das duas. Em vez de

```
strlen(strcat(s1, s2))
```
usamos
```
strlen(s1) + strlen(s2)
```
Outro caso em que a redução de força é possível é no cálculo do quadrado de um número. Se usarmos `pow(x, 2)`, em C++, o código gerado deverá calcular $e^{2.0 * \ln x}$, utilizando séries para calcular (aproximadamente) o logaritmo natural e a exponencial. Em vez disso, podemos usar `x*x`, com apenas uma multiplicação.

(É provável que as duas otimizações acima precisem ser feitas à mão.)

Otimizações de loop. Há várias otimizações que se aplicam a loops, a mais comum das quais é a transferência de computações invariantes do loop para fora dele. (Suponha que o comando a ser movido para antes do loop é `x=e;`.) Para que isto possa ser feito, é necessário verificar:

- A expressão `e` é composta apenas de constantes, ou de variáveis cujos valores não são alterados dentro do loop.
- Nenhum uso de `x`, dentro ou fora do loop deve ter acesso a um valor de `x` diferente do valor a que tinha acesso antes do original.

Em particular, o valor de `x` após a saída do loop deve ser o mesmo do programa original. Mesmo assim, como vimos anteriormente, é possível piorar alguns programas, quando o loop é executado zero vezes: o comando dentro do loop não seria executado, mas fora do loop será sempre executado uma vez.

Um exemplo de aplicação desta otimização seria (vamos supor `N>0`, para simplificar)

```
for (i=0, i<N, i++) {
   k=2*N;
   f(k*i);
}
```

que se transformaria em

```
k=2*N;
for (i=0, i<N, i++)
   f(k*i);
```

Eliminação de variáveis de indução. Outra possibilidade de otimização é a eliminação de variáveis de indução. Variáveis de indução são variáveis que assumem valores em progressão aritmética, a cada vez que o código do loop é executado. Por exemplo, em Pascal o salto da variável do `for` pode ser ±1, e uma variável que deve saltar, digamos de 2 em 2 deve ser uma variável distinta, cujos valores são controlados pelo programador. Podemos ter

```
for i=1 to N do begin
   j=2*i;
   p(j);
end;
```

As variáveis `i` e `j` são variáveis de indução. No caso, a variável `j` é a variável importante, e a variável `i` pode ser eliminada.

Podemos fazer

```
   j=0;
z: if (j>2*N) goto x;
   p(j);
   j=j+2;
   goto z;
x: ...
```

Note que este exemplo mostra também uma "redução de força": em vez das `N` multiplicações `2*i`, temos `N` adições `j+2` para calcular os valores sucessivos de `j`. A expressão constante `2*N` pode ser pré-calculada (dobrada).

Eliminação de sub-expressões comuns: o algoritmo do dag.

Vamos mostrar um algoritmo para determinar a possibilidade de eliminação de sub-expressões comuns em um trecho de código em linha reta, como um bloco básico. Durante a execução do algoritmo, vamos construir um grafo acíclico dirigido, cujos nós representam os valores distintos assumidos pelas diversas variáveis do programa, durante a execução do trecho considerado.

Um grafo acíclico dirigido (*dag*) é um grafo dirigido em que não há caminhos fechados. Note que em um grafo dirigido um caminho fechado só existe quando todas as arestas consideradas tem o mesmo sentido. Ou seja, podem existir caminhos fechados se a orientação das arestas for ignorada, como acontece no grafo não dirigido correspondente. Assim, embora tenha semelhanças com uma árvore, um dag pode ter mais de um caminho de um nó *i* para um nó *j*, e pode ter várias componentes desconexas.

Cada nó do dag corresponde ao valor de alguma variável do programa. Em princípio, nós distintos correspondem a valores que podem ser distintos durante a execução. Isto quer dizer que os nós que inicialmente correspondem a duas variáveis distintas `x` e `y` são nós distintos. Num ponto do programa após a execução de alguns comandos, digamos `x=a+b;` e `y=a+b;`, os valores de `x` e de `y` serão sempre iguais, e, assim, apenas um nó corresponderá a `x` e `y`. Dizemos que `x` e `y` *rotulam* o mesmo nó.

Por coincidência, os valores de variáveis que rotulam dois nós distintos podem ser iguais, mas isso não é garantido, e essa igualdade não será levada em consideração. Entretanto, quando duas variáveis rotulam o mesmo nó, é garantido que têm o mesmo valor. A cada momento, cada variável rotula apenas um nó, mas pode rotular mais de um nó, à medida que os comandos que mudam o valor da variável são considerados.

Podemos distinguir dois tipos de variáveis: *variáveis definidas pelo programador*, que tem valores definidos no início do trecho considerado, e cujos valores ao fim do trecho considerado serão presumivelmente usados na continuação do programa, e *variáveis temporárias*, que são usadas apenas durante a execução do trecho em questão e que portanto tem valores irrelevantes no início e no fim do trecho considerado. Estas variáveis temporárias são normalmente criadas durante a geração de código intermediário, e podem perder sua utilidade durante o processo de otimização aqui considerado, caso em que não serão necessárias no código final otimizado.

Inicialmente, cada variável definida pelo programador rotula um nó separado, que é associado ao valor inicial da variável. Os nós iniciais correspondentes a cada variável estão representados na figura em negrito. Cada vez que um comando `x=y op z;` for considerado, procuraremos um nó `n` com o operador `op`, que tenha com filhos nós rotulados por `y` e `z`. Se este nó existir, passará a ser o nó rotulado com `x`, além de outros rótulos que já existam para `n`. Se não existir, um nó será criado com essas características, e um único rótulo, `x`. Este processo vale para todos os operadores binários `op` da linguagem intermediária. Operadores unários, ternários, etc. são tratados de forma semelhante.

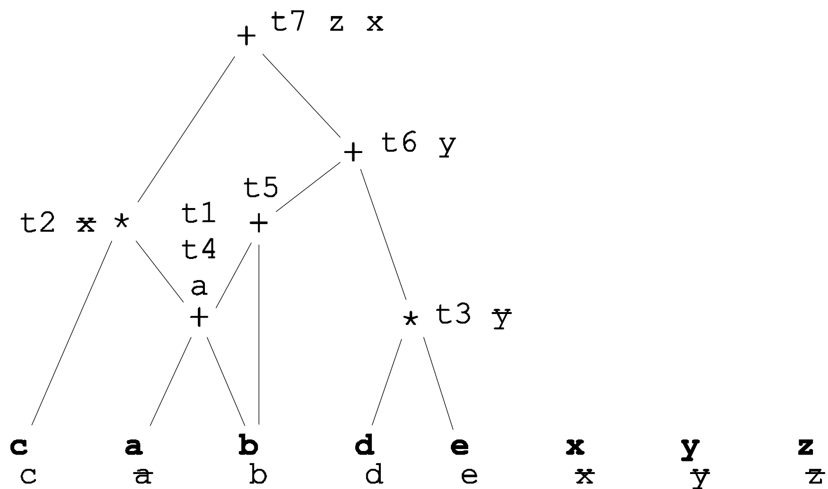
Exemplo: considere o trecho de código fonte

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

O trecho de código de 3 endereços correspondente é

```
t1=a+b;
t2=c*t1;
x=t2;
t3=d*e;
y=t3;
t4=a+b
a=t4;
t5=a+b;
t6=t5+y;
y=t6;
t7=x+y;
z=t7;
x=z;
```

A Figura abaixo mostra o dag correspondente aos comandos dados. Inicialmente, temos apenas os nós com os valores iniciais.



Os passos para construção do dag são apresentados a seguir.

0. Inicialmente são construídos os 8 nós iniciais, rotulados pelas variáveis respectivas.
1. Para o comando `t1=a+b;` criamos um nó com operador `+`, tendo como filhos os nós iniciais correspondentes a `a` e `b`. Este nó recebe o rótulo `t1`.
2. Para o comando `t2=c*t1;` criamos um nó com operador `*`, tendo como filhos o nó inicial de c, e o nó criado em (1). Este nó recebe o rótulo `t2`.
3. Para o comando `x=t2;` acrescentamos apenas ao nó criado em (2) o rótulo `x`. Este rótulo é, portanto, retirado do nó inicial de `x`, onde foi cancelado: ~~x~~.

4. Para o comando `t3=d*e;` criamos um nó com operador `*`, tendo como filhos os nós iniciais correspondentes a `d` e `e`. Este nó recebe o rótulo `t3`.
5. Para o comando `y=t3;` acrescentamos apenas ao nó criado em (4) o rótulo `y`. Este rótulo é, portanto, retirado do nó inicial de `y`, onde foi cancelado: ~~y~~.
6. Para o comando `t4=a+b;` já existe o nó com operador +, tendo como filhos os nós iniciais correspondentes a `a` e `b`, criado em (1). Este nó recebe, adicionalmente, o rótulo `t4`.
7. Para o comando `a=t4;`, acrescentamos apenas ao nó criado em (6) o rótulo `a`. Este rótulo é, portanto, retirado do nó inicial de `a`, onde foi cancelado: ~~a~~.
8. Para o comando `t5=a+b;`, o nó já existente criado em (1) não serve, porque agora é outro o nó rotulado por `a`. Portanto, um nó novo é criado, tendo como filhos o nó criado em (6) e o nó inicial de `b`. Este nó tem rótulo `t5`.
9. Para o comando `t6=t5+y;`, um novo nó é criado, com operador +, tendo como filhos o nó criado em (8) e o nó criado em (4). Este nó é rotulado com `t6`.
10. Para o comando `y=t6;`, basta mudar o rótulo `y` do nó criado em (4) para o nó criado em (9).
11. Para o comando `t7=x+y;`, um novo nó com operador +, tendo como filhos os nós criados em (2) e (9). Este nó recebe rótulo `t7`.
12. Para os comandos `z=t7;` e `x=z;`, basta acrescentar dois rótulos `z` e `x` ao nó criado em (11), cancelando os rótulos `x` e `z` anteriores.

O último passo é retirar dos nós os rótulos desnecessários: se um nó está rotulado por alguma variável declarada pelo programador, todas as variáveis temporárias são removidas; em caso contrário apenas uma variável temporária será deixada.

Nesta fase, eliminamos os rótulos `t1`, `t4`, `t6`, `t7`. Os rótulos restantes são `z` e `x` (no mesmo nó), `t2`, `t3`, `t5`, `a` e `y`.

Para gerar o código otimizado, podemos usar uma o ordem qualquer dos nós desde que um valor só seja usado após a instrução que gerou o mesmo valor. Por exemplo,

```
a=a+b;
t5=a+b;
t3=d*e;
y=t5+t3;
t2=c*a;
x=t2*y;
z=x;
```

No caso de mais de uma variável rotulando um nó (`z` e `x`, no nosso caso) apenas para uma delas o comando é gerado, sendo o valor copiado para as demais.

Para verificar que o código está correto vamos indicar para o código original e para o código otimizado os valores obtidos em cada comando, para que os valores finais de cada variável definida pelo programador sejam conferidos. Como na figura, os valores iniciais estão indicados em negrito.

Código original

```
t1=a+b;          t1=a+b
t2=c*t1;         t2=c*(a+b)
x=t2;            x=c*(a+b)
t3=d*e;          t3=d*e
y=t3;            y=d*e
t4=a+b           t4=a+b
a=t4;            a=a+b
t5=a+b;          t5=(a+b)+b
t6=t5+y;         t6=((a+b)+b)+(d*e)
y=t6;            y=((a+b)+b)+(d*e)
t7=x+y;          t7=(c*(a+b))+(((a+b)+b)+(d*e))
z=t7;            z=(c*(a+b))+(((a+b)+b)+(d*e))
x=z;             x=(c*(a+b))+(((a+b)+b)+(d*e))
```

Código otimizado

```
a=a+b;           a=a+b
t5=a+b;          t5=(a+b)+b
t3=d*e;          t3=d*e
y=t5+t3;         y=((a+b)+b)+(d*e)
t2=c*a;          t2=c*(a+b)
x=t2*y;          x=(c*(a+b))*(((a+b)+b)+(d*e))
z=x;             x=(c*(a+b))*(((a+b)+b)+(d*e))
```

Apenas os valores finais das variáveis definidas pelo programador precisam ser conferidos, uma vez que as variáveis temporárias não serão usadas posteriormente. Diz-se que estão *mortas* no final do bloco. Isso pode acontecer também com algumas das variáveis definidas pelo programador, mas consideramos neste exemplo, que todas as variáveis definidas pelo programador estão *vivas* ao final do trecho considerado.

Dependendo do número de nós esperado para o dag, a estrutura de dados usada na implementação pode ser mais ou menos complicada. Se o número esperado é grande, pode ser usada uma tabela hash para acelerar a busca dos nós. O passo básico de construção do dag consiste em procurar um nó dado pelo operador `op`, e pelos rótulos dos filhos. A tabela hash usa uma função que cria os nós numa posição dada pela função de hash aplicada à combinação do código numérico do operador com os endereços dos filhos.