Compiladores

Geração de Código

José Lucas Rangel
21 nov 00

Introdução

Enquanto a fase de análise (léxica, sintática, semântica) é essencialmente dependente da linguagem de programação em questão, a fase de geração de código é, ao contrário, dependente principalmente da máquina alvo, a máquina na qual o código gerado deverá ser executado eventualmente. Em princípio, esta não precisa ser a máquina em que a compilação se realiza, porque podemos precisar gerar código para máquinas que por seu tamanho ou por outras características não servem como plataforma para execução do compilador. Este é o caso do "software embutido" (o exemplo clássico é o "liquidificador computadorizado"): o software gerado será gravado em uma memória ROM e incluído desta maneira no aparelho que deve controlar, e não se espera que seja possível utilizar o microprocessador do dispositivo como plataforma para desenvolvimento de software.

A divisão de um compilador em "front-end" (análise) e "back-end" (geração e otimização de código) reflete estas características: em princípio, é possível escrever um front-end para cada linguagem de programação e um *back-end* para cada máquina alvo. Isto não seria razoável para linguagens e máquinas pouco convencionais, porque as linguagens intermediárias usuais poderiam ser inadequadas, mas este não é o caso que nos interessa aqui.

Um outro ponto que é importante considerar é a relação entre geração e otimização de código. Podemos ter otimização de código independente da geração de código: podemos fazer eliminação de (sub-)expressões comuns na representação intermediária, e podemos fazer otimização de "peephole" no código objeto. Entretanto, se gerarmos código sem a preocupação de eliminar código ineficiente ou desnecessário, o código gerado será muito ruim.

Um exemplo desta situação é visto a seguir.

Exemplo 1: Seja gerar código para o comando

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

em uma máquina com um registrador principal (acumulador) e instruções `Load`, `Store`, `Add`, `Sub` e `Mult`, com a semântica usual.

Quando analisamos código para este comando, identificamos o uso de várias regras, como

```
S→V=E
E→E+T
E→E-T
T→T*F
```

e geramos o seguinte código intermediário e as seguintes instruções em código objeto

```
(regra E→E+T)   t1=a+b     Load  a
                           Add   b
                           Store t1
(regra E→E-T)   t2=c+d     Load  c
                           Sub   d
                           Store t2
(regra T→T*F)   t3=t1*t2   Load  t1
                           Mult  t2
                           Store t3
(regra S→V=E)   x=t3       Load  t3
                           Store x
```

Embora a geração feita desta maneira seja correta, ela leva em conta apenas uma regra de cada vez, como é a idéia fundamental da "Tradução Dirigida pela Sintaxe", que nos permitiu centrar o processo de compilação no analisador sintático. Entretanto, se não examinarmos as combinações de regras, o software gerado pode ser de má qualidade. Por exemplo, examinemos a combinação de instruções `Store t3 Load t3`, provenientes de regras diferentes. A instrução `Load t3` é desnecessária, porque o valor armazenado em `t3` é exatamente o que está no acumulador. Mas, se esta instrução for retirada, a instrução `Load t3` também se torna inútil, porque o valor armazenado em `t3` nunca será consultado. (Outra economia é que não será necessário reservar espaço para a variável temporária `t3`.) Das 11 instruções originais temos agora apenas 9, com o mesmo efeito:

```
Load  a
Add   b
Store t1
Load  c
Sub   d
Store t2
Load  t1
Mult  t2
Store x
```

Outra transformação pode melhorar este código, de forma menos óbvia: como a multiplicação é comutativa, as instruções `Load t1 Mult t2` podem ser trocadas por `Load t2 Mult t1`, obtendo-se

```
Load  a
Add   b
Store t1
Load  c
Sub   d
Store t2
Load  t2
Mult  t1
Store x
```

onde se pode ver que a combinação `Store t2 Load t2` pode ser retirada. O resultado final é

```
Load a
Add  b
Store t1
Load c
Sub  d
Mult t1
Store x
```
com apenas 7 instruções.

Embora provavelmente este uso da identidade matemática `x*y ≡ y*x` neste exemplo não deva ter contra-indicações, isso pode não valer para identidades ou situações semelhantes. Por exemplo, em aplicações de cálculo numérico, o uso da associatividade da soma, `(x+y)+z ≡ (x+(y+z)` pode fazer com que um algoritmo convergente passe a divergir. Por essa razão, a maioria das otimizações oferecidas por um compilador pode ser desativada pelo uso de opções de compilação, através das quais o usuário especifica que formas de otimização considera razoáveis para sua aplicação.

Critérios de Otimização. Há dois critérios básicos pelos quais a qualidade do código gerado por um compilador pode ser julgada. (Supomos que o código está correto.) Esses dois critérios são tempo (velocidade de execução) e espaço (tamanho do código). Estes dois critérios podem, naturalmente, ser conflitantes.

Por exemplo, se substituirmos o código objeto correspondente a

```
for(i=1; i<=10; i++)
   f(5*i);
```
pelo código objeto correspondente a

```
f( 5);
f(10);
f(15);
f(20);
f(25);
f(30);
f(35);
f(40);
f(45);
f(50);
```
teremos um código mais longo, mas certamente mais rápido, porque dispensa as dez multiplicações, dez operações de incremento, e os onze testes de permanência do `for`.

Na prática, entretanto, há uma correlação forte entre os critérios de tempo e de espaço. Primeiro, porque remover instruções inúteis melhora o código segundo os dois critérios. Segundo, porque instruções mais longas em geral demoram mais a ser executadas. Por exemplo, uma instrução de carga de um valor a partir de um endereço em memória é mais longa e mais lenta que uma de carga de um valor a partir de um registrador. Num caso, o endereço em memória faz parte da instrução, ocupando vários bytes; no outro, o registrador pode ser indicado com apenas alguns bits. Note que a decodificação de uma instrução, preparatória para sua execução, exige que todos os bytes da instrução sejam trazidos para a CPU, o que toma mais tempo no caso de uma instrução mais longa. Isso,

além do tempo de acesso aos endereços especificados na instrução, para leitura ou escrita de dados.

Um modelo de custo para geração de código. No que se segue, vamos usar um modelo de custo das instruções, que nos permite avaliar de uma forma simplificada a qualidade do código gerado. Vamos supor uma máquina com vários registradores, r0, r1, ... e instruções como as vistas no exemplo anterior, generalizadas para permitir o tratamento dos vários registradores, e operações entre eles. As instruções que vamos usar em nossos exemplos são as seguintes:

```
                                    custo
Move  Move r r'   r = r'             1
      Move x r    x = r              2
      Move r x    r = x              2
Add   Add r r'    r = r + r'         1
      Add r x     r = r + x          2
Sub   Sub r r'    r = r - r'         1
      Sub r x     r = r - x          2
Mult  Mult r r’   r = r * r'         1
      Mult r x    r = r * x          2
```

A última coluna define o custo das instruções, sempre igual a 1 + o número de endereços de memória constantes da instrução. O valor 1 é às vezes explicado como sendo o “custo do *opcode*”.

A tabela abaixo mostra várias opções de código para o comando de atribuição do exemplo anterior.

```
Move  r0 a        Move  r0 a        Move  r0 a
Add   r0 b        Add   r0 b        Add   r0 b
Move  t1 r0       Move  t1 r0
Move  r0 c        Move  r0 c        Move  r1 c
Sub   r0 d        Sub   r0 d        Sub   r1 d
Move  t2 r0
Move  r0 t1
Mult  r0 t2       Mult  r0 t1       Mult  r0 r1
Move  t3 r0
Move  r0 t3
Move  x r0        Move  x r0        Move  x r0

Custo=22          Custo=14          Custo=11
  (a)               (b)               (c)
```

A coluna (a) mostra o código “óbvio” gerado usando as técnicas vistas de Tradução Dirigida pela Sintaxe, usando três temporárias em memória; em (b) temos o código melhorado no primeiro exemplo, que usa apenas uma temporária. A coluna (c) usa dois registradores, `r0` e `r1`, e dispensa a inversão da ordem dos argumentos na multiplicação.

Naturalmente, sempre que possível, vamos usar apenas registradores para armazenar os valores temporários, sem necessidade de usar as instruções mais caras para acessar variáveis temporárias em memória. Naturalmente, pode acontecer que o número de registradores disponíveis no hardware não seja suficiente, porque o número mínimo de registradores necessários para calcular a expressão excede o número de registradores

existentes, e neste caso deveremos complementar os registradores existentes com temporárias em memória.

Determinação do número mínimo de registradores para o cálculo de uma expressão.

Vamos agora mostrar como calcular o número mínimo de registradores necessário para o cálculo de uma expressão. Para isto algumas premissas são feitas: (1) no caso de operações comutativas é irrelevante a ordem dos argumentos e (2) existe uma maneira de inverter a ordem dos argumentos, no caso de operações não comutativas. Por exemplo, no caso da subtração, podemos supor que existe uma operação de subtração reversa (`rSub`), definida por

```
rSub r r'  r = r' - r
rSub r x   r = x - r
```

Se esta instrução não existir, podemos usar a instrução `Sub`, seguida de uma instrução de troca de sinal do registrador (`CHS`). Note que há necessidade de uma instrução adicional (`CHS`), mas o número de registradores usados não se altera.

Seja a expressão `exp = exp1 op exp2`. Suponhamos que já foi determinado que o número mínimo de registradores necessário para calcular `exp1` é `n1` (respectivamente `exp2`, `n2`). Podemos obter `n`, o número de registradores necessário para calcular `exp`, da seguinte maneira:

Se `n1=n2`, `n=1+n1` (ou, equivalentemente `1+n2`)

Se `n1>n2`, `n=n1`

Se `n1<n2`, `n=n2`

A explicação para isto é a seguinte: se `n1>n2`, calculamos inicialmente `exp1`, usando `n1` registradores; em seguida deixamos o valor de exp1 guardado em `1` registrador, e calculamos `exp2` usando `n2` registradores adicionais. Nesta segunda fase, usamos um máximo de `1+n2` registradores. Mas, como `n1>n2`, temos `1+n2`≤`n1`, e `n1` é o valor mínimo. Por outro lado, se tivermos `n1=n2`, o número de registradores na segunda fase será `1+n2`, e `1+n2>n1`, de forma que este será o número mínimo de registradores necessário. Se `n1<n2`, a situação é a mesma do caso `n1>n2`, exceto que primeiro calculamos `exp2`, e depois `exp1`.

A expressão mais simples é composta por apenas uma variável, e precisa de `0` registradores para ser calculada. Note que qualquer atribuição usa pelo menos um registrador, mesmo a atribuição mais simples, `x=y`, muito embora a expressão (uma variável) `y` possa ser calculada em `0` registradores.

Exemplo: a expressão `(a+b)*(c-d)` pode ser calculada em dois registradores:

```
                n1   n2   n
a+b             0    0    1
c+d             0    0    1
(a+b)*(c-d)     1    1    2
```

Código:

```
Move  r0 a
Add   r0 b
Move  r1 c
Sub   r1 d
Mult  r0 r1
```

Custo: 9

Exemplo: a expressão `(a+(b-c))*((e-f)+(g-h))` também pode ser calculada em dois registradores:

	n1	n2	n
b-c	0	0	1
a+(b-c)	0	1	1
e-f	0	0	1
g-h	0	0	1
(e-f)+(g-h)	1	1	2
(a+(b-c))*((e-f)+(g-h))	1	2	2

Código:

```
Move  r0 e
Sub   r0 f
Move  r1 g
Sub   r1 h
Add   r0 r1
Move  r1 b
Sub   r1 c
Add   r1 a
Mult  r1 r0
```

Custo: 16

Exemplo: a expressão `((a-b)*(c-d)+(e-f)*(g-h))` pode ser calculada em três registradores

a-b	0	0	1
c-d	0	0	1
(a-b)*(c-d)	1	1	2
e-f	0	0	1
g-h	0	0	1
(e-f)*(g-h)	1	1	2
((a-b)*(c-d)+(e-f)*(g-h))	2	2	3

Código:

```
Move  r0 a
Sub   r0 b
Move  r1 c
Sub   r1 d
Mult  r0 r1
Move  r1 e
Sub   r1 f
Move  r2 g
Sub   r2 h
Mult  r1 r2
Add   r0 r1
```

Custo:19

Exemplo: repetir o exemplo anterior, supondo que a máquina só tem dois registradores `r0` e `r1`. Como precisamos de um mínimo de três registradores, devemos liberar um registrador guardando seu valor na memória. Diz-se que foi necessário "derramar" (*spill*) o registrador.

Uma das possibilidades é salvar o valor de `(a-b)*(c-d)` de `r0` para a memória, em uma posição temporária `t1`.

```
Move  r0 a
Sub   r0 b
Move  r1 c
Sub   r1 d
Mult  r0 r1
Move  t1 r0
Move  r1 e
Sub   r1 f
Move  r0 g
Sub   r0 h
Mult  r1 r0
Move  r0 t1
Add   r0 r1
```

Este código tem custo 23. Para fazer isso de forma rápida e simples, podemos também vamos usar instruções `Push` e `Pop` para empilhar ou desempilhar na pilha de hardware. O topo da pilha de hardware substitui a temporária `t1`.

```
Move  r0 a
Sub   r0 b
Move  r1 c
Sub   r1 d
Mult  r0 r1
Push  r0
Move  r1 e
Sub   r1 f
Move  r0 g
Sub   r0 h
Mult  r1 r0
Pop   r0
Add   r0 r1
```

Este código tem o custo reduzido para 21.

Toda esta análise, entretanto, supõe que a máquina alvo possui um número suficiente de registradores "de uso geral", isto é, registradores aos quais se podem aplicar todas as instruções necessárias. As máquinas reais normalmente tem muitos registradores, mas cada um deles tem uma finalidade específica, e pode ser difícil ou impossível usá-los da forma mostrada aqui, porque nem todas as instruções podem ser aplicadas a todos os registradores. Além disso, algumas operações, como multiplicação e divisão, utilizam pares de registradores, e isto impõe restrições adicionais. Por exemplo, considere uma máquina com 16 registradores simples, indicados por `0000`, `0001`, ..., `1111`, e 8 pares de registradores duplos, `000`, `001`, ..., `111`. Cada registrador duplo `x` é composto de dois registradores simples, `x0` e `x1`, de forma que registradores simples e duplos não podem ser usados de forma independente. Por exemplo, o uso do registrador simples `1101` impede o uso do registrador duplo `110`, composto de `1100` e `1101`.

Seleção de Instrução e Alocação de Registradores.

Máquinas convencionais (CISC) costumam ter várias instruções com finalidades semelhantes. Por exemplo, vários tipos de instruções envolvem o cálculo de somas:

- uma instrução específica para soma
- uma instrução de incremento ou decremento (soma ±1)
- uma instrução com modo de endereçamento indexado ou baseado
- uma instrução de empilhar ou desempilhar (soma ou subtrai o número de bytes movidos)

Normalmente, estas instruções são usadas em situações específicas:

- a instrução de soma propriamente dita é usada para as somas indicadas pelo programador, como em `a+b`;
- instruções de incremento e decremento são usadas quando um dos operandos tem o valor ±1, como em `x-1` ou `x++`;
- os modos de endereçamento são usados para casos de variáveis indexadas, como `v[i]`, ou acesso a campos de estruturas, como em `x.a`;
- instruções de empilhar/desempilhar são usadas para passagem de parâmetros e resultados. Na realidade uma instrução de chamada de função também usa a pilha de hardware, para guardar o endereço de retorno, para uso pela instrução de retorno.

Nada nos impede, entretanto, de usar estas instruções de uma forma diferente da prevista pelo projetista do hardware. Entretanto, isto raramente acontece em geradores de código escritos à mão, ao contrário de geradores escritos por geradores de geradores de código, que funcionam de forma automática, examinando todas as possibilidades.

Assim, um dos problemas a ser resolvido na geração de código é o da *seleção da instrução*: escolher uma instrução adequada entre as diversas instruções possíveis (com diversos modos de endereçamento).

O outro problema é o da *alocação dos registradores*, que consiste na determinação das posições onde serão armazenados os dados durante a execução das instruções (os registradores). Normalmente se considera uma hierarquia de posições possíveis: os registradores propriamente ditos (na CPU), o topo da pilha e a memória em geral.

O acesso ao topo da pilha é mais rápido que o acesso a uma posição arbitrária da memória, porque não há necessidade de indicar na instrução o endereço do topo da pilha, que é mantido em um registrador específico da CPU, o apontador da pilha, ou *stack pointer*. Naturalmente, o uso de uma pilha exige que o acesso LIFO (*last in, first out*). Em particular, cada valor só pode ser usado uma vez, o que dificulta o tratamento de sub-expressões comuns, cujo valor deve ser usado mais de uma vez.

Estes dois problemas (*seleção de instrução* e *alocação de registradores*) devem em geral ser resolvidos simultaneamente, porque nem todas as instruções podem ser aplicadas a todos os registradores. Na prática, a falta de regularidade da arquitetura das máquinas obriga o programador de um gerador de código ao uso de técnicas heurísticas ou ad-hoc para resolver os dois problemas simultaneamente.

Uma heurística para geração de código de trechos em linha reta.

Vamos apresentar uma forma de geração de código que costuma funcionar bem na prática. Esta técnica será apresentada inicialmente através de um exemplo. Seja gerar código para o trecho

```
x=a-b+c;
y=d*e;
z=a+b+f;
w=x+y;
x=w;
```

na máquina usada anteriormente, usando apenas dois registradores (`r0` e `r1`). Temos como código intermediário

```
t1=a-b
t2=t1+c
x=t2
t3=d*e
y=t3
t4=a+b
t5=t4+f
z=t5
t6=x+y
w=t6
x=w
```

e, após uma fase de eliminação de sub-expressões comuns,

```
t1=a-b
x=t1+c
y=d*e
z=t1+f
w=x+y
x=w
```

Vamos anotar, durante o processo de geração de código, quais variáveis tem seu valor na memória atualizado, e quais os valores contidos nos registradores `r0` e `r1`. Inicialmente todas as variáveis tem seu valor na memória, exceto a temporária `t1`, cujo valor inicial é desconhecido, assim como o conteúdo dos registradores `r0` e `r1`. O código será gerado adiando ao máximo a passagem dos valores dos registradores para a memória, ou seja, só derramando os valores dos registradores quando isso se tornar estritamente necessário.

```
                    a b c d e f w x y z   r0={} r1={}
```

Vamos gerar código para `t1=a-b`, e escolhemos para isso um registrador vazio (`r0`).

```
Move  r0 a      a b c d e f w x y z   r0={a} r1={}
Sub   r0 b      a b c d e f w x y z   r0={t1} r1={}
```

Vamos gerar código para `x=t1+c`. Como `t1` tem um uso futuro, seu valor deve ser guardado, na memória ou em um registrador. Como `r1` está vazio, vamos usá-lo para calcular `x`. (Note que não há garantia de que o valor de `t1` não precise ser derramado posteriormente.) A ordem da operação foi invertida.

```
Move  r1 c      a b c d e f w x y z   r0={t1} r1={c}
Add   r1 r0     a b c d e f w y z     r0={t1} r1={x}
```

Observe a anotação de que o valor de x na memória está desatualizado. Vamos agora gerar código para y=d*e. Precisamos escolher um registrador para essa operação. Note que ambos os registradores tem conteúdos que nos interessam (x e t1 tem usos futuros), e que é necessário derramar um dos dois. Normalmente a preferência deve ser por manter a variável temporária no registrador, porque seu valor ao final do trecho de código considerado não tem utilidade. (Note entretanto que o valor atual de r1 não é o valor final de x.) Assim, vamos derramar r1 (para x) e calcular y em r1.

```
Move  x r1      a b c d e f w x y z   r0={t1} r1={x}
Move  r1 d      a b c d e f w x y z   r0={t1} r1={d}
Mult  r1 e      a b c d e f w x z     r0={t1} r1={y}
```

Vamos agora gerar código para z=t1+f. Como t1 está em r0, e este é o último uso de t1, vamos usar r0 para esta operação.

```
Add   r0 f      a b c d e f w x       r0={z} r1={y}
```

Vamos agora gerar código para w=x+y e x=w. Os dois devem ser considerados juntos porque apenas um valor é gerado para w e x. Escolhemos r1, porque y já está em r1. Como o valor de y é o seu valor final, que poderá usado na continuação do programa, deve ser derramado para a posição correspondente na memória.

```
Move  y r1      a b c d e f w x y     r0={z} r1={y}
Add   x r1      a b c d e f y         r0={z} r1={w, x}
```

Como os valores em w, x e z podem ser usados na continuação do programa, devem ser derramados, ao final.

```
Move  z r0      a b c d e f y z       r0={z} r1={w, x}
Move  w r1      a b c d e f w y z     r0={z} r1={w, x}
Move  x r1      a b c d e f w x y z   r0={z} r1={w, x}
```

O código final é

```
Move  r0 a
Sub   r0 b
Move  r1 c
Add   r1 r0
Move  x r1
Move  r1 d
Mult  r1 e
Add   r0 f
Move  y r1
Add   x r1
Move  z r0
Move  w r1
Move  x r1
```

Este código tem custo 25. Podemos verificar que este código poderia ser melhorado se mudássemos y=d*e para depois do último uso de t1, gerando código a partir de

```
t1=a-b
x=t1+c
z=t1+f
y=d*e
w=x+y
x=w
```

Neste caso o código gerado seria

```
Move  r0 a
Sub   r0 b       r0={t1}
Move  r1 c
Add   r1 r0      r1={x}
Add   r0 f       r0={z}
Move  z r0
Move  r0 d
Mult  r0 e       r0={y}
Move  y r1
Add   r1 r0      r1={w, x}
Move  x r1
Move  w r1
```

com custo 22.

A heurística indicada neste exemplo, entretanto, gera o código na mesma ordem do código intermediário, e não tem condições de verificar as possibilidades de reordenação do código.

Para determinar qual de duas variáveis deve ser derramada vários critérios podem ser usados, dependendo da informação disponível. Por exemplo, podemos desempatar guardando a variável cujo valor vai ser usado mais vezes no futuro, ou aquela cujo valor será usada na instrução mais próxima.

Esta heurística é basicamente a mesma indicada na função `getreg` do livro do Dragão (§9.5 – A simple code generator).