Construção de Compiladores

Otimização de código

Professor: Luciano Ferreira Silva, Dr.

Otimização de código

- O código intermediário gerado contempla cada expressão do código-fonte individualmente;
- Ao avaliar globalmente é possível observar trechos ineficientes.
- A otimização visa aplicar um conjunto de heurísticas para detectar e remover tais ineficiências;

Otimização de código

- **Existem dois tipos de técnicas de otimização:**
  - ✓ Independentes de máquina;
    - Realizadas no código intermediário;
  - ✓ Dependentes de máquina;
    - Realizadas no código em linguagem simbólica;
    - Deve-se conhecer as instruções específicas de um processador;
      - para saber quais instruções podem realizar uma mesma tarefa de forma mais eficiente;

Otimização de código

- **As técnicas de otimização analisam as instruções em blocos;**
- **O programa é representado por um grafo:**
  - ✓ Os nós são os blocos de instruções;
  - ✓ Os arcos são os possíveis caminhos de execução;
  - ✓ São analisados os fluxos de dados e controle;
- **Dentro de cada bloco analisa-se a definição e o uso da variáveis;**
  - ✓ Para descobrir as possibilidades de utilização de técnicas de otimização associadas aos seus valores;
  - ✓ Estas estratégias são denominadas estratégias de otimização local.

Otimização de código

- **Considere o trecho de código intermediário já visto:**

```
_t1 := 4 * i
_t2 := a[_t1]
_t3 := 4 * i
_t4 := b[_t3]
_t5 := t2 + _t4
_t6 := 4 * i
s[_t6] := _t5
```

- **_t1, _t3 e _t6 têm sempre o mesmo valor;**
  - ✓ Portanto, _**t3** e _**t6** podem ser substituídas por _t1;

```
_t1 := 4 * i
_t2 := a[_t1]
_t4 := b[_t1]
_t5 := t2 + _t4
s[_t1] := _t5
```

Otimização de código

- **Esse é um exemplo de uma das possíveis técnicas de otimização, a eliminação de subexpressões comuns;**
- **Outra heurística é a eliminação de código redundante.**
  - ✓ O objetivo dessa estratégia é detectar expressões repetidas;
  - ✓ Por exemplo:

```
le := id
...
le := id
```

  - ✓ Existem três possibilidades:
  - ✓ ***id*** é alterado entre as instruções: não remover;
  - ✓ ***le*** é alterado entre as instruções: não remover;
  - ✓ Caso contrário, remove-se a última instrução;

Otimização de código

- Outra técnica que exige a análise de fluxo de dados é a propagação de cópias;
- Explora a igualdade de valores entre variáveis distintas;

```
le1 := id
...
le2 := le1
```

- Se os valores de id e le1 não forem alterados entre as duas instruções:

```
le2 := id
```

Otimização de código

- Heurísticas de otimização relacionadas à análise do fluxo de controle do programa:

```
        <a>
        goto _L1
_L1:    <b>
```

- A instrução de desvio é desnecessária pode ser removida:

```
        <a>
_L1:    <b>
```

- Caso o rótulo _L1 não esteja referenciado por outra instrução do programa também pode ser eliminado.

Otimização de código

- Com a análise do fluxo de controle também é possível aplicar a estratégia de eliminação de código não-alcançável ou "código morto":

```
        ...
        goto _L1
        <a>
_L1:    ...
```

- Perceba, toda este bloco de comando pode ser eliminado.

Otimização de código

- O uso de propriedades algébricas é outra estratégia de otimização usualmente aplicada;
- Por exemplo:

Substituir	Por
x + 0	x
0 + x	x
x – 0	x
x * 1	x
1 * x	x
x / 1	x

Otimização de código

- **Diversas oportunidades de otimização estão associadas à análise de comandos iterativos;**
  - ✓ Uma estratégia é a movimentação de código;
    - Aplicada quando um cálculo realizado dentro do laço envolve valores invariantes na iteração;
- **Considere o seguinte código em C++:**

```
while (i < 10*j){
        a[i] = i + 2*j;
        i++;
}
```

Otimização de código

- Código intermediário sem otimização:

```
_t1 := 10 * j
_L1: if i >= _t1 goto _L2
       _t2 := 2 * j
       _t3 := i + _t2
       _t4 := 4 * i
       a[_t4] := _t3
       i := i + 1
       goto _L1
_L2: ...
```

- Código intermediário com otimização:

```
_t1 := 10 * j
_t2 := 2 * j
_L1: if i >= _t1 goto _L2
       _t3 := i + _t2
       _t4 := 4 * i
       a[_t4] := _t3
       i := i + 1
       goto _L1
_L2: ...
```

Geração de código em linguagem simbólica

- A última etapa do compilador propriamente dito é a geração do código em linguagem simbólica;
- Uma vez que esse código seja gerado, outro programa – o montador – será responsável pela tradução para o código objeto, em formato de linguagem de máquina;
- O código intermediário limita o número de operadores e os tipos de instruções para facilitar a produção de código em linguagem simbólica.

Geração de código em linguagem simbólica

- **A abordagem mais simples para essa etapa é ter;**
  - ✓ para cada instrução do formato intermediário;
    - um gabarito com a correspondente seqüência de instruções em linguagem simbólica do processador-alvo;
- **Uma das características importantes a considerar do processador é o número de operandos com que a instrução opera;**
  - ✓ Há uma classificação de processadores de acordo com o número de operandos especificados em uma operação binária na linguagem simbólica.

Geração de código em linguagem simbólica

Máquinas de três endereços:

- As instruções correspondentes a operações binárias explicitam os endereços dos dois operandos de entrada, assim como o endereço no qual o resultado será armazenado;
- Por exemplo, ***z = a + b*** pode ser codificada por:

```
ADD a, b, z        ou        ADD z, a, b
```

- Tipicamente, trabalha-se com registradores.

Geração de código em linguagem simbólica

Máquinas de dois endereços:

- As instruções correspondentes a operações binárias explicitam apenas os endereços dos dois operandos de entrada;
- O resultado é implicitamente assumido como sendo o mesmo do primeiro operado;
- Por exemplo: ***ADD a, b*** produz o resultado de ***a = a + b***;
- Tipicamente, neste caso, o primeiro operando não é uma variável do programa, mas sim um registrador do processador;
  - Desse modo, o valor original da variável não é alterado;

Geração de código em linguagem simbólica

- Uma instrução de transferência entre memória e registrador, como ***MOVE***, é utilizada;
- Por exemplo a operação ***z = a + b*** seria traduzida para:

```
MOVE a, R0
ADD  R0, b
MOVE R0, z
```

- Nesse caso, foi assumido que o primeiro argumento da instrução ***MOVE*** é a origem da transferência e o segundo argumento o seu destino;
  - Dependendo do processador, a forma de codificação pode ser ao contrário.

Geração de código em linguagem simbólica

Máquinas de um endereço:

- As operações binárias especificam apenas o endereço de um operando, usualmente o segundo;
- O endereço do primeiro operando e do resultado são implicitamente assumidos como sendo um registrador especial, o acumulador;
- Tipicamente a transferência:
  - da memória para o acumulador é realizada pela instrução ***LOAD***;
  - do acumulador para a memória é feita pela instrução ***STORE***;
- Para ***z = a + b*** tem-se:

```
LOAD    a
ADD     b
STORE   Z
```

Geração de código em linguagem simbólica

Máquinas de zero endereços:

- A operação binária não explicita nenhum endereço;
- O processador assume que os operandos são retirados de uma pilha;
  - Que pode ser de registradores ou na memória;
- O resultado também é armazenado na pilha;
- Tipicamente os operandos são transferidos:
  - da memória para pilha com uma operação ***PUSH***;
  - da pilha para memória com uma instrução ***POP***;
- Para ***z = a + b*** tem-se:

```
PUSH  a
PUSH  b
ADD
POP   z
```

Geração de código em linguagem simbólica

- O código gerado da tradução pode também causar redundâncias.
- Por exemplo, considere o código intermediário:

```
a := b + c
d := a + e
```

- Para uma máquina de dois endereços:

```
MOVE b, R0
ADD  R0, c
MOVE R0, a
MOVE a, R0
ADD  R0, e
MOVE R0, d
```

- Perceba, a quarta instrução é desnecessária poderia ser removida.

Geração de código em linguagem simbólica

- Mais um exemplo de tradução:

```
L0: if I = 100 goto L1
        I := J * I
        goto L0
L1: . . .
```

- Tradução para linguagem simbólica (arquitetura x86):

```
L0 MOV AX, I
   CMP AX, 100
   JGE L1
   MOV BX, J
   MUL BX
   MOV I, AX
   JMP L0
L1
```