Compiladores (CC3001)
Aula 15: Análise de Fluxo e Otimização de código

Mário Florido

DCC/FCUP

2024

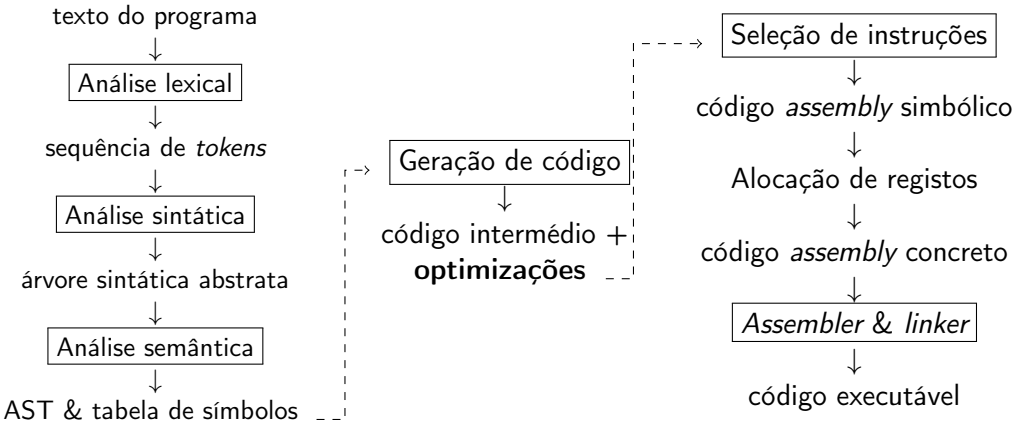
[dcc] FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Otimizações: visão global

Análises de fluxo

- *Reaching Definitions*
- *Available Assignments*
- *Liveness*

Re-lembrar: fases dum compilador



Otimizações: visão global

Análises de fluxo

- *Reaching Definitions*
- *Available Assignments*
- *Liveness*

- A tradução para código intermédio é dirigida pela sintaxe
- O código de cada parte da AST é gerado de forma independente das restantes
- Isso gera sequências de instruções sub-ótimas
- Podemos utilizar passagens suplementares de **otimização** para substituir essas sequências por outras mais eficientes
- Não é possível um compilador produzir sempre o código ótimo (limites da computabilidade)
- Mas é possivel um compilador mais sofisticado produzir código mais eficiente do que um compilador mais simples

Em que fase efetuar otimizações?

- Podemos efetuar otimizações no código intermédio ou no código *assembly* (ou ambos)
- Vantagens de otimizar código intermédio:
  - O código intermédio é mais simples que o *assembly*
  - As otimizações em código intermédio beneficiam qualquer *backend*
- Mas certas otimizações de baixo-nível só são possíveis no *assembly* (por exemplo: usar instruções *assembly* especializadas)
- Nesta aula vamos ver otimizações de código intermédio
- As técnicas para otimização de *assembly* são semelhantes

Como efetuar otimizações?

- Condição de segurança: só fazer otimizações que não mudem o comportamento observável do programa
- Como responder à pergunta "é seguro fazer uma certa otimização aqui"?
- Podemos usar *análises de fluxo de dados* (*dataflow*) para obter matematicamente informação sobre o comportamento do programa
  - As análises calculam conjuntos que aproximam a semântica do programa
  - Por causa da incomputabilidade, a resposta será sempre aproximada
  - Em certos casos isso impede o compilador de fazer a otimização (aproximação conservadora)

Onde efetuar otimizações?

Ao nível do programa completo:

- Problema: o custo de otimização cresce muito com o tamanho do programa
- Compilação de módulos separados obrigaria a fazer otimização na fase de *link*
- Não é normalmente efetuado por compilador genéricos

Ao nível das funções ou procedimentos:

- O tamanho de cada função é limitado mesmo em programas grandes
- Oferece boas oportunidades para otimizações
- Efetuado por todos os compiladores otimizadores (GCC, GHC, etc)

Exemplos de otimizações

Register alocation Manter dois no mesmo registo temporários que não interferem (aula passada)

Common subexpression elimination Se uma expressão for calculada múltiplas vezes, eliminar uma das computações

Dead-code elimination Eliminar uma computação que nunca será usada

Constant folding Se os operandos de uma operação são constantes, efetuar a operação em tempo de compilação

Esta aula

Otimizações: visão global

Análises de fluxo

- *Reaching Definitions*
- *Available Assignments*
- *Liveness*

- Análises intraprocedimentais: aplicadas a cada função separadamente
- Grafo de fluxo: nós (instruções) numerados de 1 até n
- Conjuntos *in* e *out* de informação à *entrada* e *saída* de cada instrução
- Exprimimos o fluxo de informação à custa de conjuntos *gen* e *kill*
- Podemos calcular *in* e *out* de forma iterativa até estabilizar
- *Forward analyses*: informação flui de *out* de um nó para *in* dos sucessores
- *Backward analyses*: informação flui de *in* de um nó para *out* dos predecessores

Esta aula

Otimizações: visão global

Análises de fluxo

- *Reaching Definitions*
- *Available Assignments*
- *Liveness*

- Calcular quais as definições que podem chegar a cada ponto do programa
- A instrução i da forma $t := e$ chega a um ponto j se existir um caminho no grafo de controlo de fluxo de i até j sem uma redefinição $t := e'$
- *in* e *out* são conjuntos de índices das instruções

Conjuntos *gen* e *kill* para cada instrução:

instrução i	$gen[i]$	$kill[i]$
$t := a \oplus b$	$\{i\}$	$defs(t) \setminus \{i\}$
$t := a$	$\{i\}$	$defs(t) \setminus \{i\}$
$t := M[a]$	$\{i\}$	$defs(t) \setminus \{i\}$
$M[a] := b$	$\emptyset$	$\emptyset$
$t :=$ `CALL` $f(args)$	$\{i\}$	$defs(t) \setminus \{i\}$
`COND` c L_1 L_2	$\emptyset$	$\emptyset$
`JUMP` L	$\emptyset$	$\emptyset$
`LABEL` L	$\emptyset$	$\emptyset$

$defs(t)$ conjunto dos índices das instruções da forma $t := e$

Equações de fluxo:

$$in[i] = \bigcup_{j \in pred[i]} out[j] \tag{1}$$

$$out[i] = gen[i] \cup (in[i] \setminus kill[i]) \tag{2}$$

- Uma definição *chega* a um nó sse *sai* de algum dos dos antecessores
- A informação flui de $in[i]$ para $out[i]$ conforme os conjuntos $gen[i]$ e $kill[i]$
- Podemos usar as equações para calcular *in* e *out* por iteração começando com $in[i] = out[i] = \emptyset$

```
1. a := 5
2. c := 1
3. LABEL L1
4. COND c>a L2 L3
5. LABEL L3
6. c := c+c
7. JUMP L1
8. LABEL L2
9. a := c-a
10. c := 0
```

				*Iter*1		*Iter*2	
i	$pred[i]$	$gen[i]$	$kill[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$
1		1	9		1		1
2	1	2	6, 10	1	1, 2	1	1, 2
3	2, 7			1, 2	1, 2	1, 2, 6	1, 2, 6
4	3			1, 2	1, 2	1, 2, 6	1, 2, 6
5	4			1, 2	1, 2	1, 2, 6	1, 2, 6
6	5	6	2, 10	1, 2	1, 6	1, 2, 6	1, 6
7	6			1, 6	1, 6	1, 6	1, 6
8	4			1, 2	1, 2	1, 2, 6	1, 2, 6
9	8	9	1	1, 2	2, 9	1, 2, 6	2, 6, 9
10	9	10	2, 6	2, 9	9, 10	2, 6, 9	9, 10

Iteração 3 é identica à iteração 2 (estabiliza).

Constant Propagation substituir variáveis que são constantes pelo seu valor

Exemplo: apenas uma definição de *a* (`a:=5`) chega à instrução 4, logo podemos substituir

```
4. COND c>a L2 L3  ⟶  4. COND c>5 L2 L3
```

Podemos ainda combinar esta otimização com:

Constant folding substituir operações aritméticas entre constantes pelo seu resultado

```
a := 5          a := 5          a := 5
b := 3    ⟶    b := 3    ⟶    b := 3
t := a*b        t := 5*3        t := 15
```

Otimizações: visão global

Análises de fluxo

- *Reaching Definitions*
- *Available Assignments*
- *Liveness*

- **Common Subexpression Elimination**: Simplificar ocorrências de expressões repetidas em código intermédio
- Exemplo resultante da compilação de `a[i] = a[i] + 19`

```
t1 := 4*i              t1 := 4*i              t1 := 4*i
t2 := a+t1             t2 := a+t1             t2 := a+t1
t3 := M[t2]            t3 := M[t2]            t3 := M[t2]
t4 := t3+19     ⟶      t4 := t3+19     ⟶      t4 := t3+19
t5 := 4*i              t5 := t1               t5 := t1
t6 := a+t5             t6 := a+t5             t6 := t2
M[t6] := t4            M[t6] := t4            M[t6] := t4
```

- Para efetuar esta otimização necessitamos de saber quais as atribuições que estão disponíveis em cada ponto do programa
- Vamos efetuar uma análise de *Available Assignments*
- Calcula conjuntos de atribuições $x := e$ (pares de variáveis e expressões)

$in[i]$ atribuições à entrada da instrução i

$out[i]$ atribuições à saida da instrução i

$gen[i]$ atribuições geradas pela instrução i

$kill[i]$ atribuições invalidadas pela instrução i

$pred[i]$ instruções que antecedem i

Conjuntos *gen* e *kill* para cada instrução ($x \neq y$ e $x \neq z$):

instrução i	$gen[i]$	$kill[i]$
$x := y \oplus z$	$\{x := y \oplus z\}$	$assg(x)$
$x := x \oplus z$	$\emptyset$	$assg(x)$
$x := k$	$\{x := k\}$	$assg(x)$
$x := M[y]$	$\{x := M[y]\}$	$assg(x)$
$M[x] := y$	$\emptyset$	*loads*
$x :=$ `CALL` $f(args)$	$\emptyset$	$assg(x) \cup$ *loads*
`JUMP` *label*	$\emptyset$	$\emptyset$
`COND` c $label_1$ $label_2$	$\emptyset$	$\emptyset$

$assg(x)$ atribuições em que x ocorre do lado esquerdo ou direito

loads atribuições da forma $z := M[\cdot]$ para alguma variável z

Exemplo

```
1. i:=0
2. a:=n*3
3. COND i<a loop end
4. LABEL loop
5. b:=i*4
6. c:=p+b
7. d:=M[c]
8. e:=d*2
9. f:=i*4
10. g:=p+f
11. M[g]:=e
12. i:=i+1
13. a:=n*3
14. COND i<a loop end
15. LABEL end
16. RETURN p
```

i	$pred[i]$	$gen[i]$	$kill[i]$
1		1	1, 5, 9, 12
2	1	2	2
3	2		
4	3, 14		
5	4	5	5, 6
6	5	6	6, 7
7	6	7	7, 8
8	7	8	8
9	8	9	9, 10
10	9	10	10
11	10		7
12	11		1, 5, 9, 12
13	12	2	2
14	13		
15	3, 14		
16	15		

$$in[i] = \bigcap_{j \in pred[i]} out[j] \qquad (i \neq 1) \tag{3}$$

$$out[i] = gen[i] \cup (in[i] \setminus kill[i]) \tag{4}$$

- *Forward analysis*: a informação flui de *in* para *out*
- Junção usando $\bigcap$ em vez de $\bigcup$: um *assignment* está disponível à entrada de i se estiver disponível à saida de *todos* os predecessores de i
- Iteração de ponto-fixo: começamos com $in[1] = \emptyset$, $out[1] = \mathcal{A}$ e $in[i] = out[i] = \mathcal{A}$ para $i \neq 1$, onde $\mathcal{A}$ é o conjunto de todas as atribuições do programa.

Exemplo

	Inicialização		Iter 1		Iter 2	
i	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$
1		$1,2,5,6,7,8,9,10$		1		1
2	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	1	$1,2$	1	$1,2$
3	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2$	$1,2$	$1,2$	$1,2$
4	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2$	$1,2$	2	2
5	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2$	$1,2,5$	2	$2,5$
6	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5$	$1,2,5,6$	$2,5$	$2,5,6$
7	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6$	$1,2,5,6,7$	$2,5,6$	$2,5,6,7$
8	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,7$	$1,2,5,6,7,8$	$2,5,6,7$	$2,5,6,7,8$
9	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8$	$1,2,5,6,7,8,9$	$2,5,6,7,8$	$2,5,6,7,8,9$
10	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9$	$1,2,5,6,7,8,9,10$	$2,5,6,7,8,9$	$2,5,6,7,8,9,10$
11	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,8,9,10$	$2,5,6,7,8,9,10$	$2,5,6,8,9,10$
12	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,8,9,10$	$2,6,8,10$	$2,5,6,8,9,10$	$2,6,8,10$
13	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$2,6,8,10$	$2,6,8,10$	$2,6,8,10$	$2,6,8,10$
14	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$2,6,8,10$	$2,6,8,10$	$2,6,8,10$	$2,6,8,10$
15	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	2	2	2	2
16	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	2	2	2	2

Iteração 3 é idêntica à iteração 2 (estabiliza).

Se uma instrução i é da forma $x := e$ para alguma expressão e e $in[i]$ contém uma atribuição $y := e$, podemos substituir $x := e$ por $x := y$.

No exemplo:

- na instrução 9 (`f:=i*4`) temos a atribuição 5 disponível (`b:=i*4`) pelo que podemos substuir 9 por `f := b`
- na instrução 13 (`a:=n*3`) temos a atribuição 2 disponível (`a:=n*3`), pelo que podemos substuir 13 por a:=a e eliminá-la (no-op)
- ainda não eliminamos a expressão `p+f` na instrução 10 apesar de ter o mesmo valor que atribuição 2 disponível (`c:=p+b`)
- podemos substituir `f` por `b` e repetir a análise

Usando *Available Assignments* (cont.)


```
1. i:=0
2. a:=n*3
3. COND i<a loop end
4. LABEL loop
5. b:=i*4
6. c:=p+b
7. d:=M[c]
8. e:=d*2
9. f:=i*4
10. g:=p+f
11. M[g]:=e
12. i:=i+1
13. a:=n*3
14. COND i<a loop end
15. LABEL end
16. RETURN p
```

$\longrightarrow$

```
1. i:=0
2. a:=n*3
3. COND i<a loop end
4. LABEL loop
5. b:=i*4
6. c:=p+b
7. d:=M[c]
8. e:=d*2
9. f:=b
10. g:=p+f
11. M[g]:=e
12. i:=i+1
13. a:=a
14. COND i<a loop end
15. LABEL end
16. RETURN p
```

Otimizações: visão global

Análises de fluxo

- *Reaching Definitions*
- *Available Assignments*
- *Liveness*

- Já vimos que a análise de *Liveness* pode ser usada para a alocação de registos
- Vamos ver que pode também ser usada para algumas otimizações
- **Dead-code Elimination**: se a variável x não está viva depois de uma atribuição

$$x := e$$

  e e é uma *expressão sem efeitos colaterais*, então podemos eliminar essa atribuição
- Expressões com efeitos colaterais: ler da memória; chamar outras funções
- Expressões sem efeitos colaterais: constantes, variáveis, operações aritméticas[1]

[1] Excepto se houver verificação de *overflow* ou divisão por 0.

Análise intra-procedimental sobre uma função com n instruções numeradas de 1 até n.

$succ[i]$ O conjunto das instruções que podem ser executadas imediatamente após a instrução i.

$gen[i]$ O conjunto de variáveis cujos valores são usados na instrução i.

$kill[i]$ O conjunto das variáveis que são escritas pela instrução i.

$in[i]$ O conjunto das variáveis que estão vivas à entrada da instrução i.

$out[i]$ O conjunto das variáveis que estão vivas à saida da instrução i.

Equações de fluxo:

$$in[i] = gen[i] \cup (out[i] \setminus kill[i]) \tag{5}$$

$$out[i] = \bigcup_{j \in succ[i]} in[j] \tag{6}$$

Iteração de ponto-fixo:

1. Inicializar $in[i] := out[i] := \emptyset$
2. Calcular por ordem $out[n], in[n], out[n-1], in[n-1], \ldots out[1], in[1]$ usando as equações acima
3. Repetir o ponto 2 até estabilizar