

Análise de Fluxo de Dados

Available Expressions

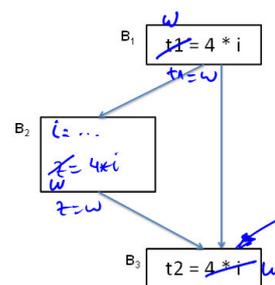
- Expressão disponível:
 - $x+y$ está disponível em p se:
 - **todo caminho** do nó inicial até p calcula $x+y$
 - após a última computação de $x+y$, nem x nem y sofrem atribuições



Available Expressions

Instruções	Expressões disponíveis
\emptyset	\emptyset
$a = b + c$	\emptyset
$b = a - d$	$b+c$
$c = b + c$	$a-d$
$d = a - d$	$a-d$
\emptyset	\emptyset

Available Expressions



Available Expressions

- **Kill:**
 - Um bloco B mata, ou pode matar, $x+y$ se ele atribui a x e/ou y , e **não recomputa $x+y$**
- **Gen:**
 - Um bloco B gera $x+y$ se ele certamente computa $x+y$, e não redefine x ou y .

Equações da DFA

- Computamos *gen* e *kill* para cada B como visto anteriormente
- Temos: $out[ENTRY] = \emptyset$

Para todo bloco B diferente de $ENTRY$

$$out[B] = gen(B) \cup (in(B) - Kill(B))$$

$$in[B] = \bigcap_{p \in pred B} out(p)$$

Equações da DFA

- Computamos *gen* e *kill* para cada B como visto anteriormente
- Temos: $out[ENTRY] = \emptyset$

Para todo bloco B diferente de ENTRY

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

$$in[B] = \bigcap_{P \in \text{predecessores de } B} out[P]$$

Available Expressions

$$out[ENTRY] = \emptyset$$

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

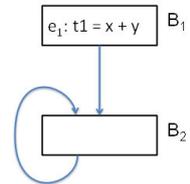
$$in[B] = \bigcap_{P \in \text{predecessores de } B} out[P]$$

$$in[B_1] =$$

$$out[B_1] =$$

$$in[B_2] =$$

$$out[B_2] =$$



Available Expressions

$$out[ENTRY] = \emptyset$$

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

$$in[B] = \bigcap_{P \in \text{predecessores de } B} out[P]$$

$$in[B_1] = \emptyset$$

$$out[B_1] = \{e_1\}$$

$$in[B_2] = out[B_1] \cap out[B_2]$$

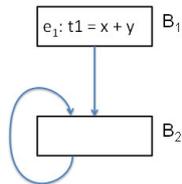
$$= \{e_1\} \cap \emptyset$$

$$= \emptyset$$

$$out[B_2] = gen[B_2] \cup (in[B_2] - kill[B_2])$$

$$= \emptyset \cup (\emptyset - \emptyset)$$

$$= \emptyset$$



Available Expressions

$$out[ENTRY] = \emptyset$$

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

$$in[B] = \bigcap_{P \in \text{predecessores de } B} out[P]$$

// Iniciar out[B] com todas as expressões

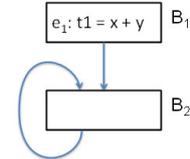
$$out[B_1] = \{e_1\}$$

$$out[B_2] = \{e_1\}$$

$$in[B_1] =$$

$$in[B_2] =$$

$$out[B_2] =$$



Diferenças para Reaching Defs

- O operador de confluência é intersecção
 - Tem que vir por todos os caminhos
- Estimativa inicial é grande
 - Intersecção vai diminuindo os conjuntos até chegar ao maior ponto fixo

Solução Iterativa

$$OUT[ENTRY] = gen[ENTRY]; IN[ENTRY] = \{\}$$

for (each basic block B other than ENTRY)

$$OUT[B] = U - e_kill[B] // U = \text{Todas as expressões}$$

while (changes to any OUT occur)

for (each basic block B other than ENTRY) {

$$in[B] = \bigcap_{P \in \text{predecessores de } B} out[P]$$

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

}

Exemplo

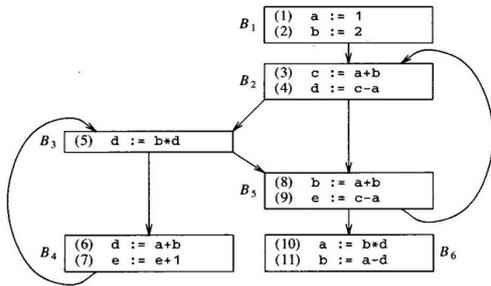


Fig. 10.74. Flow graph.

Live-Variable Analysis

- Linguagem intermediária
 - Gerada pelo *front-end* considerando número infinito de registradores para temporários
- Máquinas reais têm um número finito de registradores
 - 32 é um número típico para máquinas RISC
- Dois valores temporários podem ocupar o mesmo registrador se não estão “em uso” ao mesmo tempo
 - Muitos temporários podem caber em poucos registradores
 - Os que não couberem vão para a memória (*spill*)

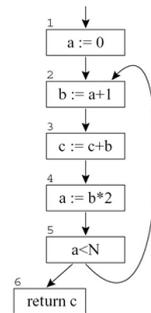
Live-Variable Analysis

- Dizemos que uma variável está *viva* em um ponto p se ela pode vir a ser usada no futuro em um caminho a partir de p .
- O compilador analisa a RI para saber quais variáveis estão *vivas* ao mesmo tempo.
- Esta tarefa então, é conhecida como *live-variable analysis*, ou *liveness analysis* (análise de longevidade).

Exemplo

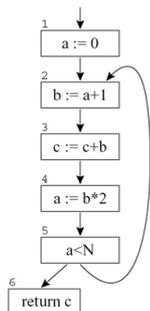
```

a ← 0
L1 : b ← a + 1
    c ← c + b
    a ← b * 2
    if a < N goto L1
    return c
    
```



Exemplo

Em quais pontos do CFG a variável b está viva?



Exemplo

- Quantos registradores precisamos para alocar todas as variáveis do programa anterior?

Live-Variable Analysis

- $IN[B]$: conjunto de variáveis vivas no início de B
- $OUT[B]$: conjunto de variáveis vivas no final de B
- def_B : conjunto de variáveis atribuídas em B antes de qualquer uso daquela variável em B.
- use_B : conjunto de variáveis utilizadas em B antes de qualquer atribuição à variável em B.

B_2 <pre> b = a + 1 c = c + b a = b + 2 if a < N goto B₂ </pre>	$def_{B_2} =$ $use_{B_2} =$
---	------------------------------------

Computando Liveness

1. Se a variável v está em use_B , então v é *live-in* em B e *pertence a* $in[B]$.
2. Se a variável v é *live-in* no bloco B, então ela é *live-out* para todo bloco P que precede B.
3. Se a variável v é *live-out* no bloco B, e não está em def_B , então v é também *live-in* em B.

Equações da DFA

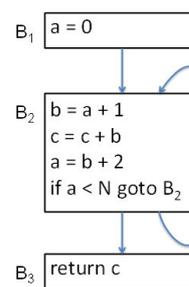
- Computamos *use (gen)* e *def (kill)* para cada B como visto anteriormente
- Neste caso, $IN[B]$ é definido em função de $OUT[B]$
- Temos: $in[EXIT] = \emptyset$

Para todo bloco B diferente de EXIT

$$in[B] = use_B \cup (out[B] - def_B)$$

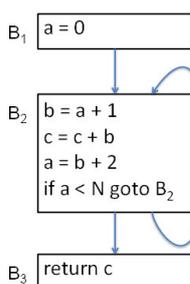
$$out[B] = \bigcup_{S \in \text{sucessores de } B} in[S]$$

Exemplo



$in[B_1] =$	$def_{B_1} =$
$use_{B_1} =$	$out[B_1] =$
$in[B_2] =$	$def_{B_2} =$
$use_{B_2} =$	$out[B_2] =$
$in[B_3] =$	$def_{B_3} =$
$use_{B_3} =$	$out[B_3] =$

Exemplo



$in[B_1] = \{c\}$
$def_{B_1} = \{a\}$
$use_{B_1} = \{\}$
$out[B_1] = \{a, c\}$
$in[B_2] = \{a, c\}$
$def_{B_2} = \{b\}$
$use_{B_2} = \{a, c\}$
$out[B_2] = \{a, c\}$
$in[B_3] = \{c\}$
$def_{B_3} = \{\}$
$use_{B_3} = \{c\}$
$out[B_3] = \{\}$

Diferenças para as anteriores

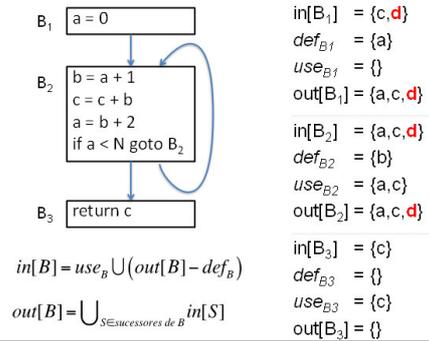
- Direção da análise
 - O conjunto $IN[B]$ é definido em função de $OUT[B]$
 - Backward: $in = f_s (out)$

Solução Iterativa

```

IN[EXIT] = {}; // Conjunto vazio
for (each basic block B other than EXIT)
  IN[B] = {};
while (changes to any IN occur)
  for (each basic block B other than EXIT) {
    out[B] =  $\bigcup_{S \in \text{sucessores de } B} \text{in}[S]$ 
    in[B] = useB  $\cup$  (out[B] - defB)
  }
  
```

Esta é uma solução válida?



Complexidade do Algoritmo

- Qual é a complexidade do algoritmo?

```

IN[EXIT] = {}; // Conjunto vazio
for (each basic block B other than EXIT)
  IN[B] = {};
while (changes to any IN occur)
  for (each basic block B other than EXIT) {
    out[B] =  $\bigcup_{S \in \text{sucessores de } B} \text{in}[S]$ 
    in[B] = useB  $\cup$  (out[B] - defB)
  }
  
```

Def-Use Chain

- Liga cada *definição* aos *usos* que alcança
- Pode ser calculada a partir das *use-def chains* (mais comum). *Use-def chains* são calculadas a partir de *reaching definitions*.
- Pode ser definida como um DFA com as mesmas equações de *live-variable analysis*

Def-Use Chain

- OUT[B]: usos alcançáveis a partir do final de B
- use_B: usos $u_i(x)$ que aparecem antes de qualquer definição não ambígua à variável x.
- def_B: todos os usos $u_i(x)$ para as variáveis x que forem definidas no bloco básico.
- Equações e algoritmo são idênticos.

Aula 6

Grafo de Interferência

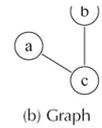
- A informação de liveness é usada para otimização
 - Alocação de registradores
- Interferência: ocorre quando a e b não podem ocupar o mesmo registrador
 - *Live ranges* com sobreposição
 - a não pode ser alocada a r1

Grafo de Interferência

- Representação:

	a	b	c
a			x
b			x
c	x	x	

(a) Matrix



Grafo de Interferência

- MOVE: é importante não criar falsas interferências entre a fonte e destino

```

t ← s           (copy)
  ⋮
x ← ... s ...   (use of s)
  ⋮
y ← ... t ...   (use of t)
    
```

- S e t estariam vivas após a instrução de cópia
- Devemos aproveitar o mesmo registrador

Grafo de Interferência

1. Definição de a que não seja move:
 1. Live-out = b1,..., bj
 1. Adicione as arestas (a, b1),..., (a, bj).

2. Moves a ← c:
 1. Live-out = b1,..., bj
 1. Adicione as arestas (a, b1),..., (a, bj) para os bi's que não são o mesmo que c