Análise de Fluxo de Dados

Sandro Rigo
sandro@ic.unicamp.br

Introdução

- Otimizações:
  - São transformações para ganho de eficiência.
  - Não podem alterar a saída do programa.
- Exemplos:
  - *Dead Code Elimination*: Apaga uma computação cujo resultado nunca será usado
  - *Register Allocation*: Reaproveitamento de registradores
  - *Common-subexpression Elimination*: Elimina expressões redundantes, que computam o mesmo valor.
  - *Constant Folding*: Se os operandos são constantes, calcule a expressão em tempo de compilação.

Introdução

- Essas transformações são feitas com base em informações coletadas do programa.
- Esse é o trabalho da análise de fluxo de dados.
- *Intraprocedural global optimization*
  - Interna a um procedimento ou função
  - Engloba todos os blocos básicos

Análise do Fluxo de Dados

- Idéia básica
  - Coletar informações sobre o programa (*data-flow information*) para cada ponto do programa.
- Informações interessantes:
  - Definições alcançantes (*reaching definitions*)
  - Variáveis vivas (*Live variables*)
  - Expressões disponíveis (*Available expressions*)

Pontos e Caminhos

- Para cada instrução do programa existe um ponto p_i antes e um ponto p_{i+1} depois da instrução.
- Um caminho de p_1 até p_n é uma sequência de pontos $p_1, p_2, \ldots, p_n$ tal que para cada i entre 1 e n-1:
  - p_i é um ponto antes de uma instrução e p_{i+1} é um ponto depois de uma instrução; ou
  - p_i é o ponto no fim de um bloco básico e p_{i+1} é um ponto no início de outro bloco básico.

Pontos e Caminhos

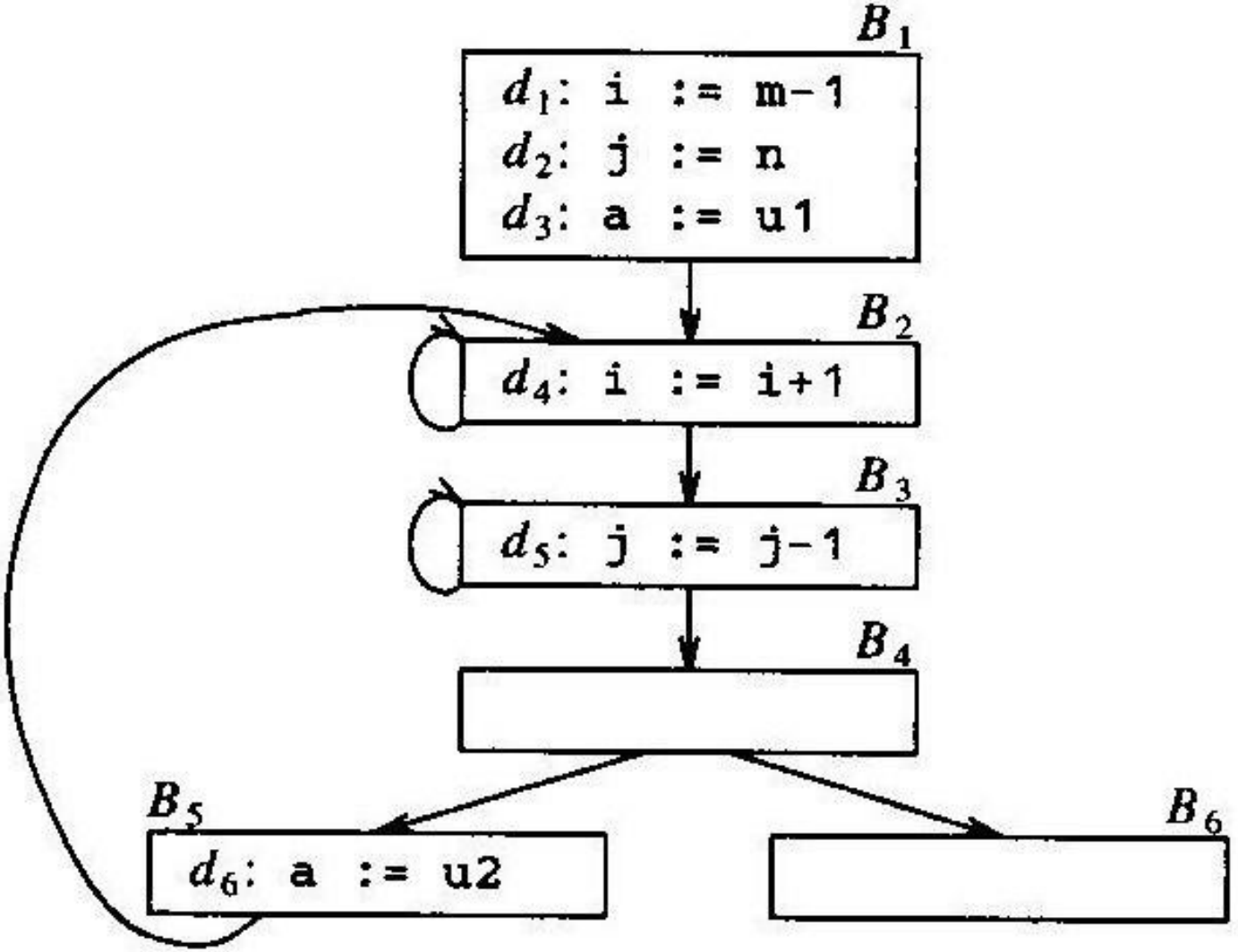


Fig. 10.19. A flow graph.

Ref: Cap 10 do Livro do Aho et al.

Análise do Fluxo de Dados

- Como coletar informações?
  - Resolvendo um sistema de **equações de fluxo de dados**.
  - Cada instrução *s* possui um conjunto de informações antes (IN[s]) e após (OUT[s]) a mesma.
  - Uma instrução *s* define restrições f_s entre os conjuntos IN[*s*] e OUT[*s*]: Ex: OUT[*s*] = f_s (IN[*s*])
  - Arestas do CFG da instrução s_i para s_j, definem restrições entre os conjuntos OUT[s_i] e IN[s_j].

Análise do Fluxo de Dados

- Exemplo: definições alcançantes no bloco básico
  - OUT[s] = *gen*[s] U (IN[s] – *kill*[s])

a = b + c

b = a + d

a = a + b

Análise do Fluxo de Dados

- As equações podem mudar de acordo com a análise:
  - As noções de *gen* e *kill* dependem da informação desejada
  - Direção da análise:
    - *Forward*: $out = f_s (in)$
    - *Backward*: $in = f_s (out)$
  - Chamadas de procedimentos, atribuição a ponteiros e a vetores
    - não vamos considerá-las no primeiro momento

Análise do Fluxo de Dados

- *Reaching Definitions* (definições alcançantes)
- *Liveness Analysis*
- *Available Expressions*

Reaching Definitions

- Principal uso:
  - Dada uma variável x em um certo ponto do programa
    - Inferimos que o valor de x é limitado a um determinado grupo de possibilidades

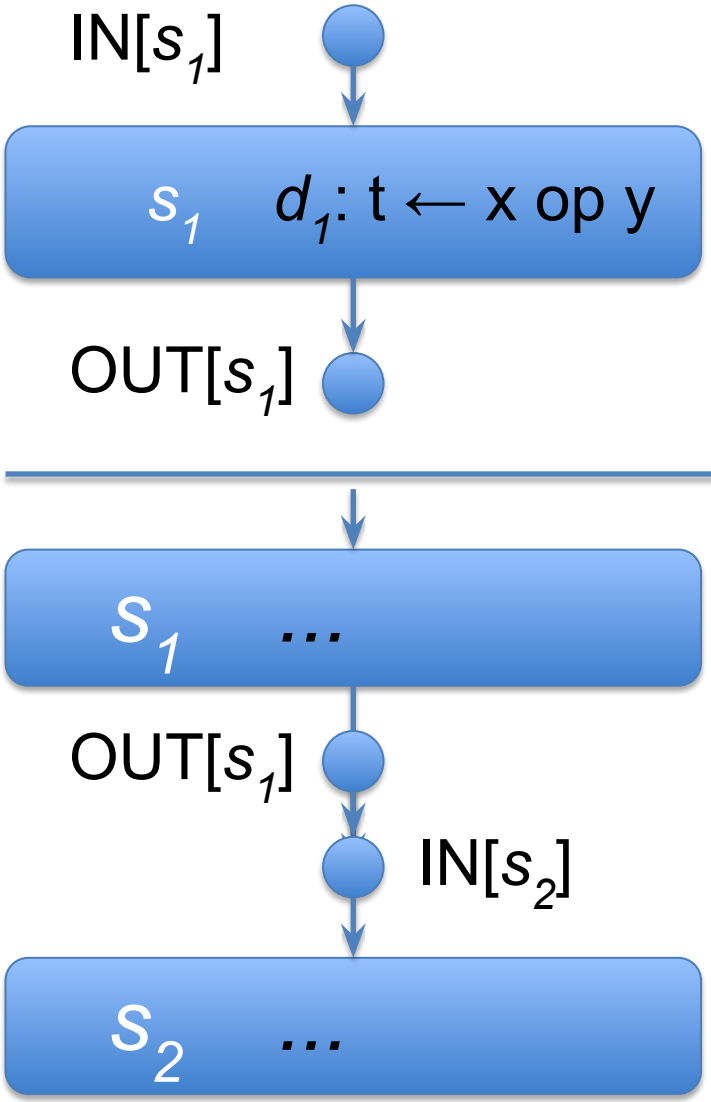
Reaching Definitions

- Definição não ambígua de t:
  - *d*: t := a op b
  - *d*: t := M[a]
- *d* alcança um uso na instrução *u* se:
  - Se existe um caminho no CFG da instrução que define *d* para *u*
  - Esse caminho não contém outra definição não ambígua de *t*
- Definição ambígua
  - Uma sentença que pode ou não atribuir um valor a *t*
    - CALL
    - Atribuição a ponteiros
    - Instruções com predicados

Reaching Definitions

- Criamos identificadores para as definições
  - d_1: t ← x op y
    - Gera d_1
    - Mata todas as outras definições de t, pois não alcançam o final dessa instrução
- defs(t) ou D_t: conjunto de todas as definições de t no programa (ou função)

Reaching Definitions



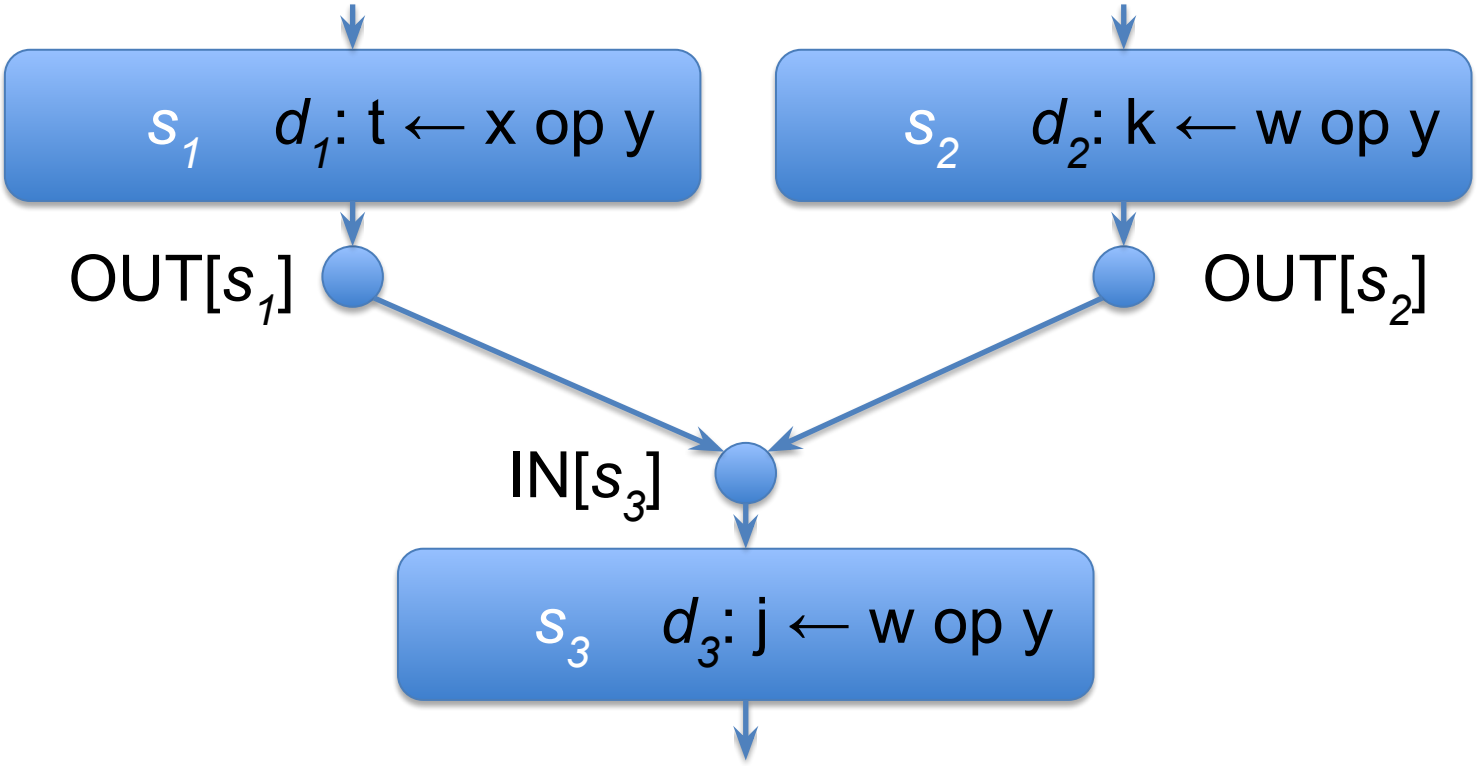
$OUT[s_1] = gen_s \cup (IN[s_1] - kill_s)$

$gen_s = \{d_1\}$

$kill_s = \{D_t - d_1\}$

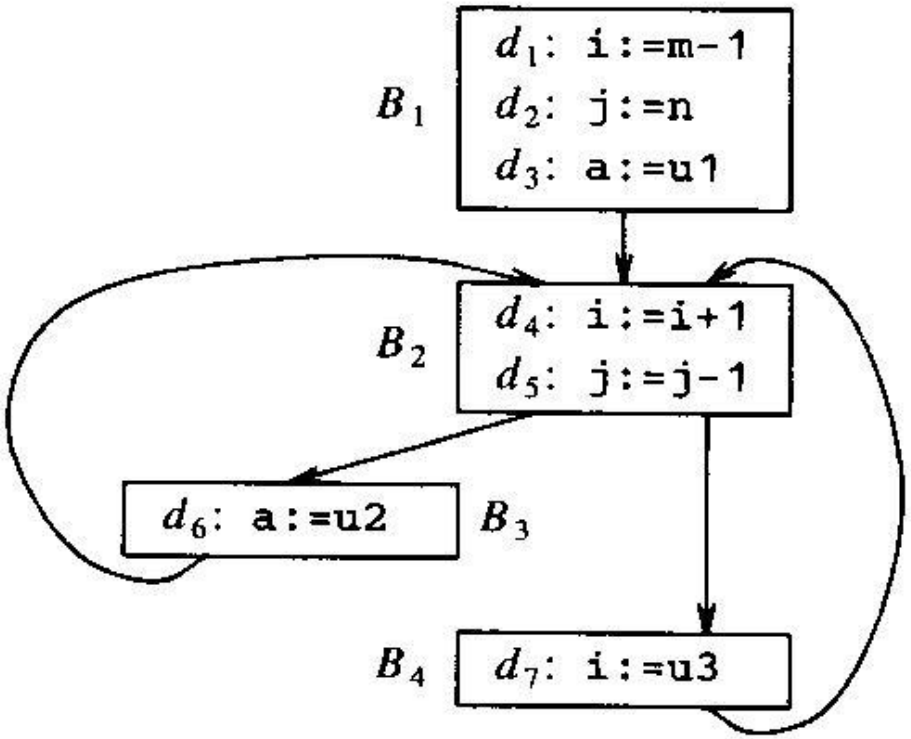
$IN[s_2] = OUT[s_1]$

Reaching Definitions



Exemplo

Definições	
i	
j	
a	

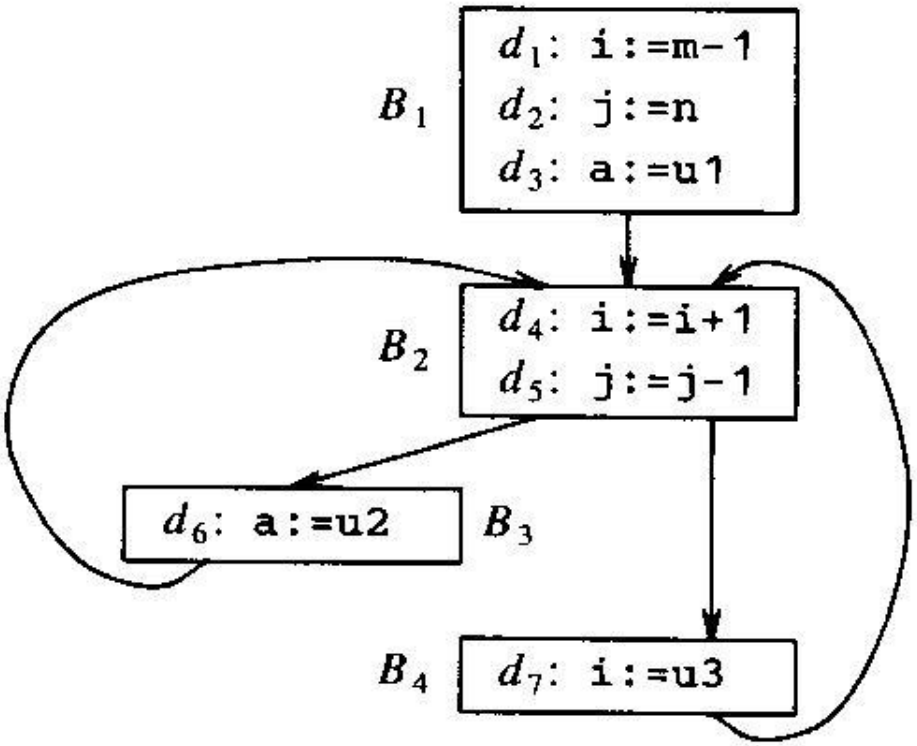


s	gen_s	$kill_s$	IN[s]	OUT[s]
1				
2				
3				
4				
5				
6				
7				

Reaching Definitions

- Agrupando equações por blocos básicos
  - Em vez de OUT[*s*] = f_s (IN[*s*])
  - Fazemos: OUT[B] = f_B (IN[B])
- Reaching definitions em blocos básicos
  - OUT[B] = gen_B U (IN[B] – $kill_B$)
  - IN[B] = $U_{\text{P é predecessor de B}}$ OUT[P]
  - $kill_B = kill_{s1} \cup kill_{s2} \ldots \cup kill_{sn}$
  - $gen_B = gen_{sn} \cup (gen_{sn-1} - kill_n) \cup \ldots$

Exemplo



Definições	
i	d_1, d_4, d_7
j	d_2, d_5
a	d_3, d_6

B	gen_B	$kill_B$	IN[B]	OUT[B]
1				
2				
3				
4				

Solução Iterativa

OUT[ENTRY] = {}

for (each basic block *B* other than *ENTRY*)

  OUT[*B*] = {}

while (changes to any OUT occur)

  for (each basic block *B* other than *ENTRY*) {

    IN[*B*] = U OUT[P];
    P a predecessor of *B*

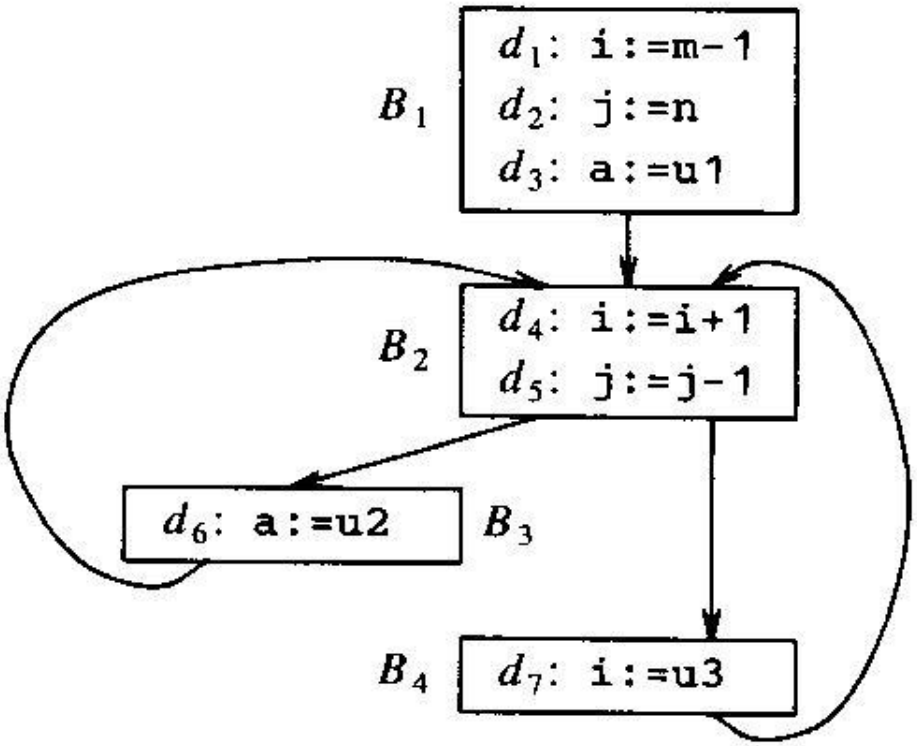
Observações

- O algoritmo propaga as definições
  - Até onde elas podem chegar sem serem mortas
  - Até que não haja mais modificações (todas as restrições são satisfeitas)
- O algoritmo sempre termina:
  - OUT[B] nunca diminui de tamanho
  - o número de definições é finito
  - se OUT não muda, IN não muda nó próximo passo
  - Limitante superior para número de iterações
    - Número de nós no CFG
    - Pode ser melhorado de acordo com a ordem de avaliação dos nós

Use-def Chains

- Armazenam a informação de *reaching definitions*
- São listas para cada uso de uma variável contendo as definições que alcançam esse uso
  - Considere variável *a* no bloco B
    - Se B não contém definições de a, *ud-chain* é o conjunto de definições de a em in[B]
    - Se B contém definições de a, então a ud-chain é a última dessas definições, antes do uso.

Exemplo



B	IN[B]	OUT[B]
1		d1, d2, d3
2	d1, d2, d3, d6	d3, d4, d5, d6
3	d3, d4, d5, d6	d4, d5, d6
4	d3, d4, d5, d6	d3, d5, d6, d7

ud-chain	