Análise de Fluxo de Dados

Sandro Rigo
sandro@ic.unicamp.br

Introdução

- Otimização
  - 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: Se uma expressão é computada mais de uma vez, elimine uma das computações
  - Constant Folding: Se os operandos são constantes, calcule a expressão em tempo de compilação

Introdução

- Essas transformações são feitas com base em informações coletas 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

Introdução

- Idéia básica
  - Atravesse o grafo de fluxo do programa coletando informações sobre a execução
  - Conservativamente!
  - Modifique o programa para torná-lo mais eficiente em algum aspecto:
    - Desempenho
    - Tamanho
- Análises são descritas através de equações de fluxo de dados:
  - out[S] = gen[S] U (in[S] – kill[S])

Introdução

- As equações podem mudar de acordo com a análise:
  - As noções de gen e kill dependem da informação desejada
  - Pode seguir o fluxo de controle ou não
    - Forward
    - Backward
  - Chamadas de procedimentos, atribuição a ponteiros e a arrays
    - não vamos considerá-las no primeiro momento

Introdução

- Veremos análises baseadas no CFG de quádruplas:
  - a ← b op c é representada como (a, b, c, op)
- Liveness Analysis
- Reaching Definitions
- Available Expressions

Pontos e Caminhos

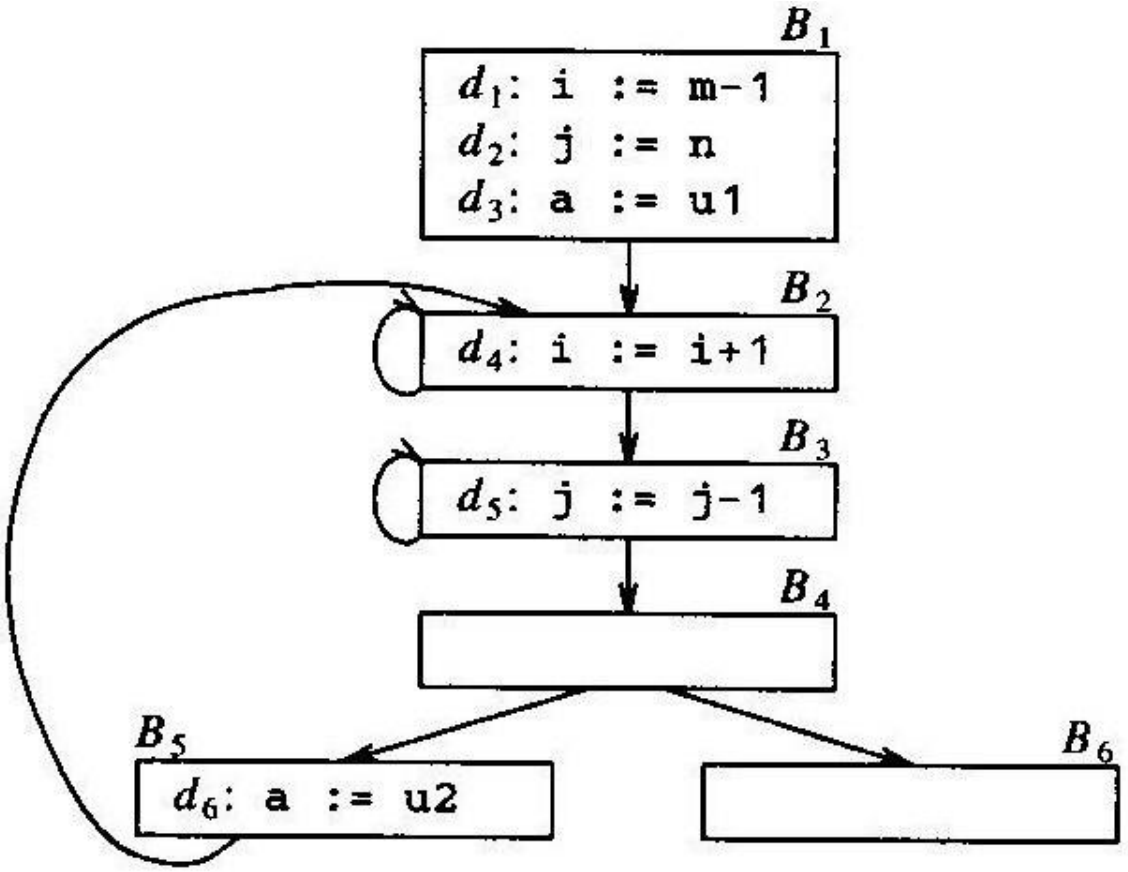


Fig. 10.19. A flow graph.

Reaching Definitions

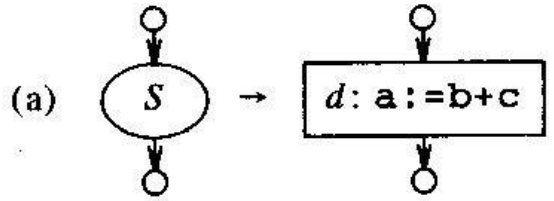
- Definição não ambígua de t:
  - d: t := a op b
  - d: t := M[a]
- d alcança um uso na sentença u se:
  - Se existe um caminho no CFG de d para u
  - Esse caminho não contém outra definição não ambígua de t
- Definição ambígua
  - Uma sentença que pode ou não atribuir um valor a t
    - CALL
    - Atribuição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

Reaching Definitions

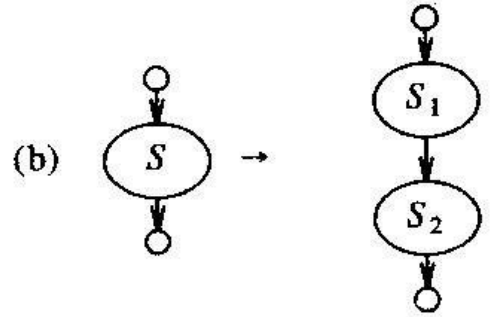
- Criamos IDs para as definições
  - d1: t ← x op y
    - Gera d1
    - Mata todas as outras definições de t, pois não alcançam o final dessa instrução
- defs(t) ou D_t: conjunto de todas as definições de t

Reaching Definitions

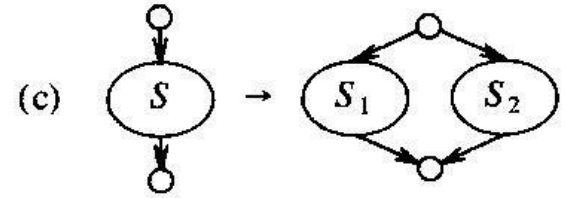
- Principal uso:
  - Dada uma variável x em um certo ponto do programa
    - Inferimos que o valor de x é limitado a um determinado grupo de possibilidades



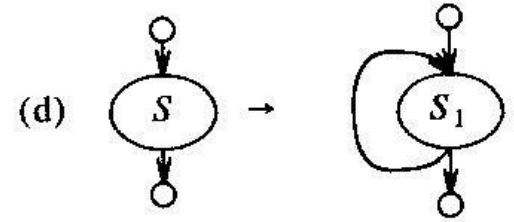
$$gen[S] = \{d\}$$
$$kill[S] = D_a - \{d\}$$
$$out[S] = gen[S] \cup (in[S] - kill[S])$$



$$gen[S] = gen[S_2] \cup (gen[S_1] - kill[S_2])$$
$$kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$$
$$in[S_1] = in[S]$$
$$in[S_2] = out[S_1]$$
$$out[S] = out[S_2]$$



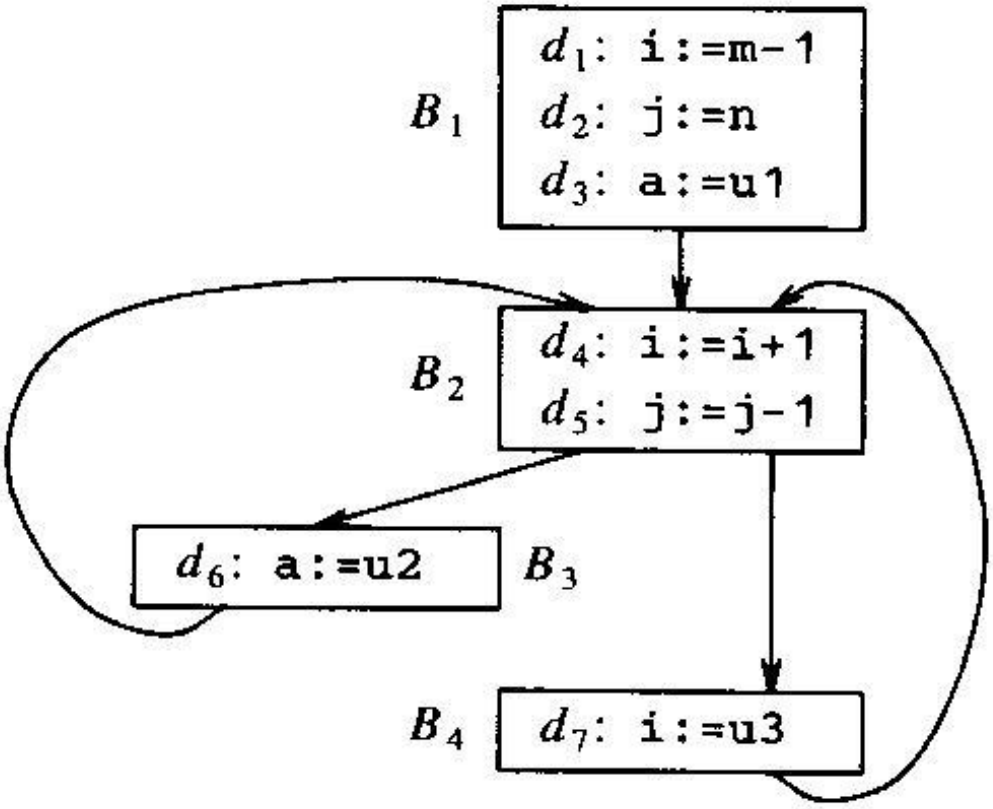
$$gen[S] = gen[S_1] \cup gen[S_2]$$
$$kill[S] = kill[S_1] \cap kill[S_2]$$
$$in[S_1] = in[S]$$
$$in[S_2] = in[S]$$
$$out[S] = out[S_1] \cup out[S_2]$$



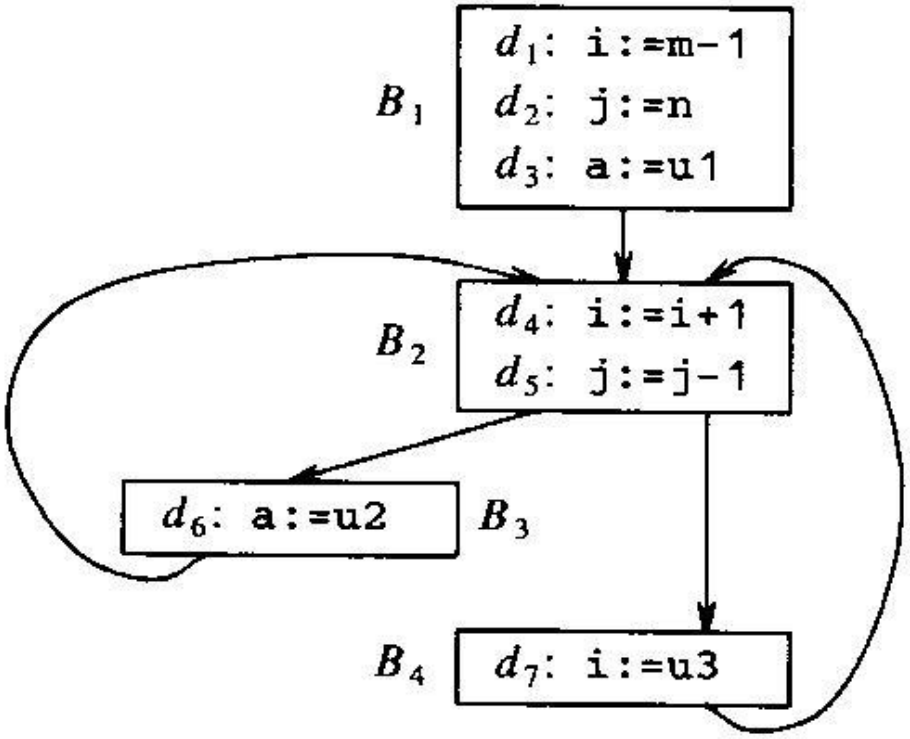
$$gen[S] = gen[S_1]$$
$$kill[S] = kill[S_1]$$
$$in[S_1] = in[S] \cup gen[S_1]$$
$$out[S] = out[S_1]$$

Fig. 10.21. Data-flow equations for reaching definitions.

Exemplo



Exemplo - Resposta



$$gen[B_1] = \{d_1, d_2, d_3\}$$
$$kill[B_1] = \{d_4, d_5, d_6, d_7\}$$

$$gen[B_2] = \{d_4, d_5\}$$
$$kill[B_2] = \{d_1, d_2, d_7\}$$

$$gen[B_3] = \{d_6\}$$
$$kill[B_3] = \{d_3\}$$

$$gen[B_4] = \{d_7\}$$
$$kill[B_4] = \{d_1, d_4\}$$

Fig. 10.27. Flow graph for illustrating reaching definitions.

Equações de DFA

- Vendo B como uma sequência de uma ou mais sentenças
  - Como vimos, podemos definir
    - In[B], out[B], gen[B], kill[B]
  - Computamos gen e kill para cada B como visto anteriormente
- Temos:

$$in[B] = \bigcup_{P \in Pred(B)} out[P]$$

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

Solução Iterativa

```
    /* initialize out on the assumption in[B] = ∅ for all B */
(1) for each block B do out[B] := gen[B];
(2) change := true;          /* to get the while-loop going */
(3) while change do begin
(4)     change := false;
(5)     for each block B do begin
(6)         in[B] := ⋃ out[P];
                     P a predecessor of B
(7)         oldout := out[B];
(8)         out[B] := gen[B] ∪ (in[B] − kill[B]);
(9)         if out[B] ≠ oldout then change := true
        end
    end
```

Fig. 10.26. Algorithm to compute *in* and *out*.

Observações

- O algoritmo propaga as definições
  - Até onde elas podem chegar sem serem mortas
  - “Simula” todos os caminhos de execução
- O algoritmo sempre pára:
  - 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 no. de iterações
    - Número de nós no CFG
    - Pode ser melhorado de acordo com a ordem de avaliação dos nós

Exemplo

- Reaching Definitions para fig 10.27

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 conjto. 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.

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
- 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.

Gen e Kill ???

STATEMENTS	AVAILABLE EXPRESSIONS
	..... none
a := b+c	
	..... only b+c
b := a-d	
	..... only a-d
c := b+c	
	..... only a-d
d := a-d	
	..... none

Fig. 10.30. Computation of available expressions.

Equações de DFA

- Computamos gen e kill para cada B como visto anteriormente
- Temos:

$$in[B] = \bigcap_{P \in \mathrm{Pred}(B)} out[P] \quad \textit{para B não inicial}$$

$$in[B1] = \varnothing$$

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

Diferenças para Reaching Defs

- O *in* do nó inicial é sempre vazio
  - Nada está disponível antes do início do programa
- O operador de confluência é intersecção
  - Tem que vir por todos os caminhos
- Estimativa inicial é muito grande
  - Intersecção vai diminuindo os conjuntos a chegar ao maior ponto fixo

Algoritmo

Algorithm 10.3. Available expressions.

Input. A flow graph G with $e\_kill[B]$ and $e\_gen[B]$ computed for each block B. The initial block is B_1.

Output. The set $in[B]$ for each block B.

Method. Execute the algorithm of Fig. 10.32. The explanation of the steps is similar to that for Fig. 10.26. □

```
in[B1] := ∅;
out[B1] := e_gen[B1];  /* in and out never change for the initial node, B1 */
for B ≠ B1 do out[B] := U − e_kill[B];  /* initial estimate is too large */
change := true;
while change do begin
        change := false;
        for B ≠ B1 do begin
                in[B] := ⋂ out[P];
                         P a predecessor of B
                oldout := out[B];
                out[B] := e_gen[B] ∪ (in[B] − e_kill[B]);
                if out[B] ≠ oldout then change := true
        end
end
```

Fig. 10.32. Available expressions computation.

Exemplo

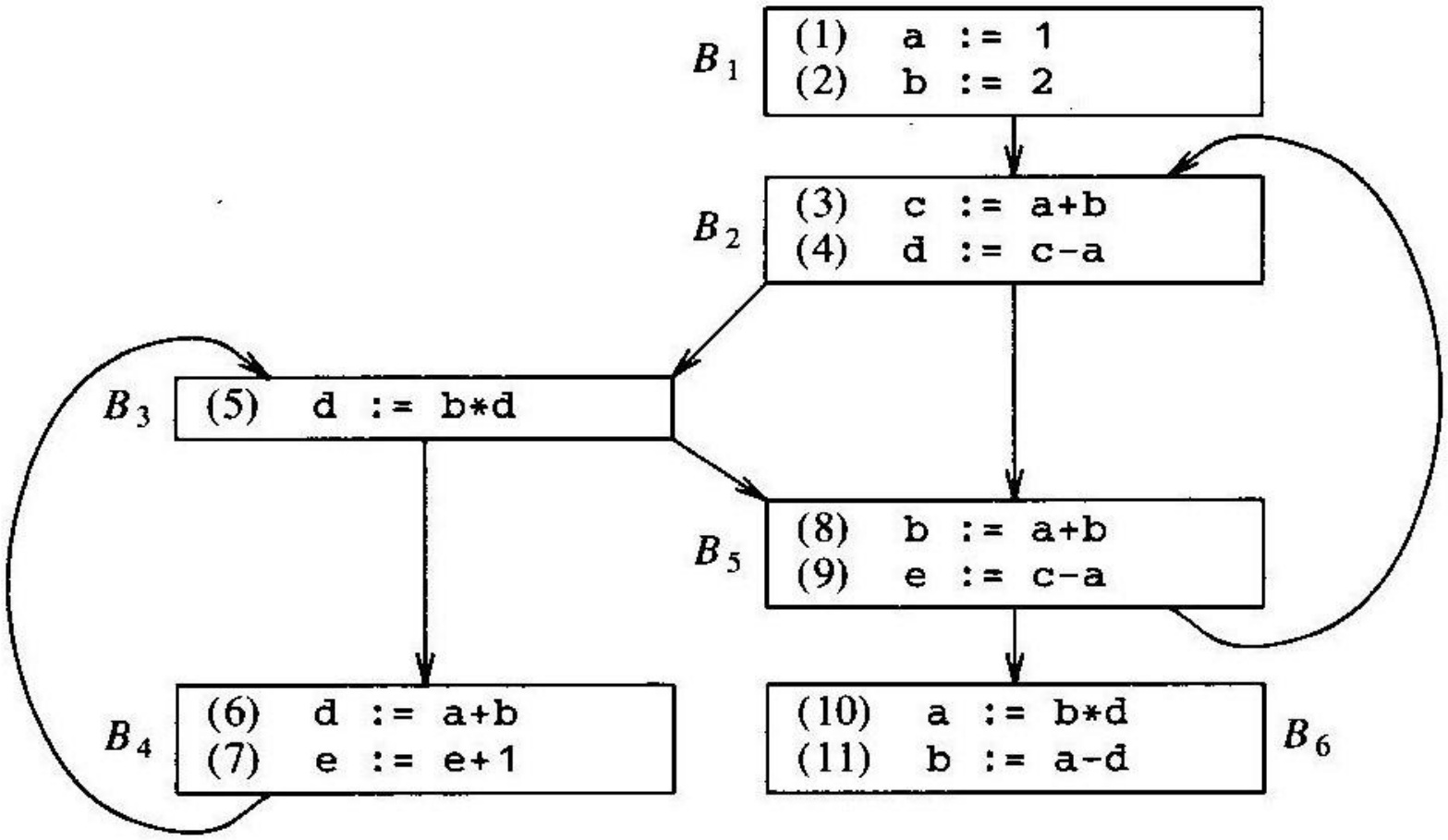


Fig. 10.74. Flow graph.