Análise de Fluxo de Dados

Guido Araújo
guido@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
  - 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
  - Register Allocation: Reaproveitamento de registradores

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
- Maioria das análises podem ser descrita através de equações de fluxo de dados:
  - Ex.: Análise de Longevidade (Cap 10)

Exemplos de Otimizações de Código Básicas

Otimizações

- Melhorar o algoritmo é tarefa do programador
- O compilador pode ser útil para
  - Aplicar transformações que tornam o código gerado mais eficiente
  - Deixa o programador livre para escrever um código limpo

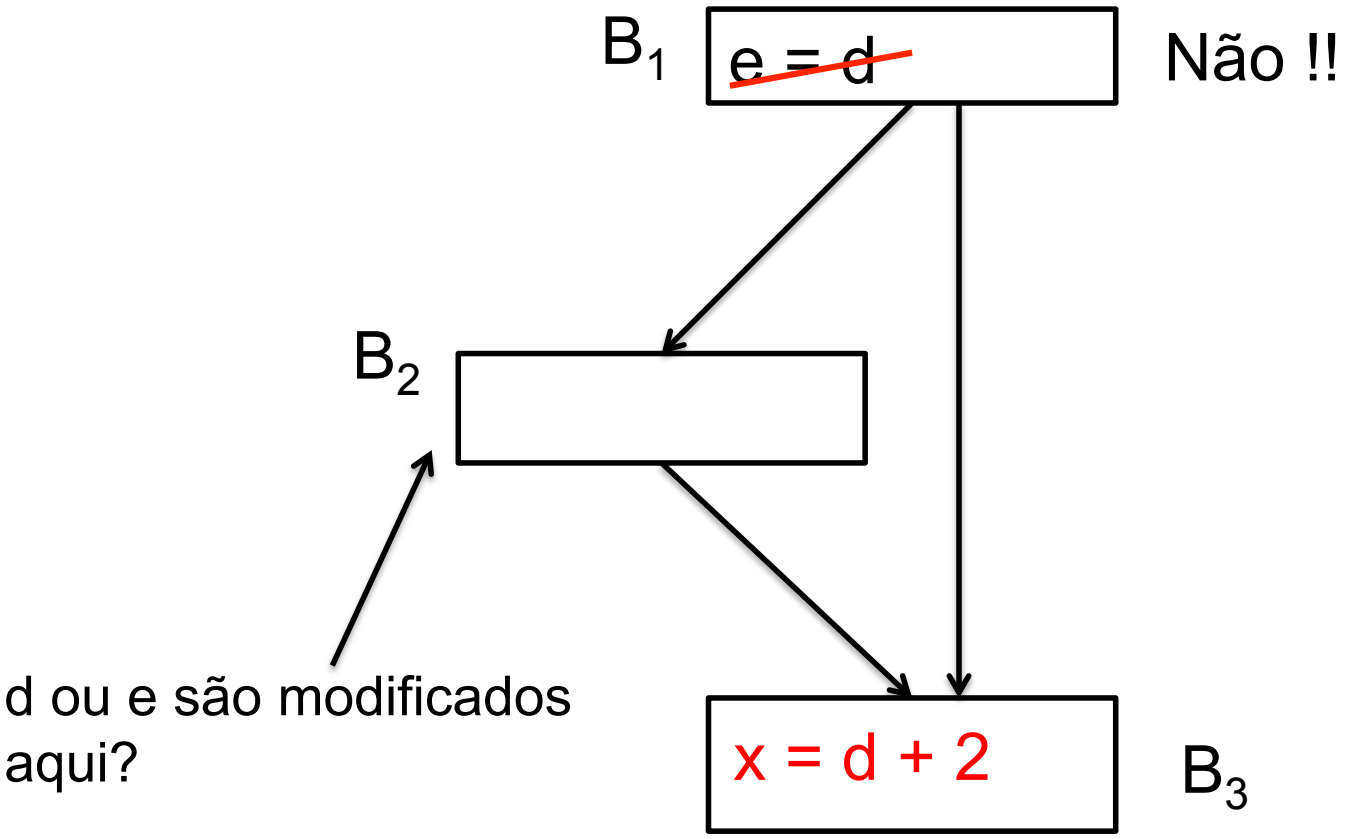
Introdução

- Usar as informações coletadas pelas análises
- Tornar o código mais eficiente
- Vamos começar olhando:
  - Dead Code Elimination
  - Constant Propagation
  - Copy Propagation
  - CSE

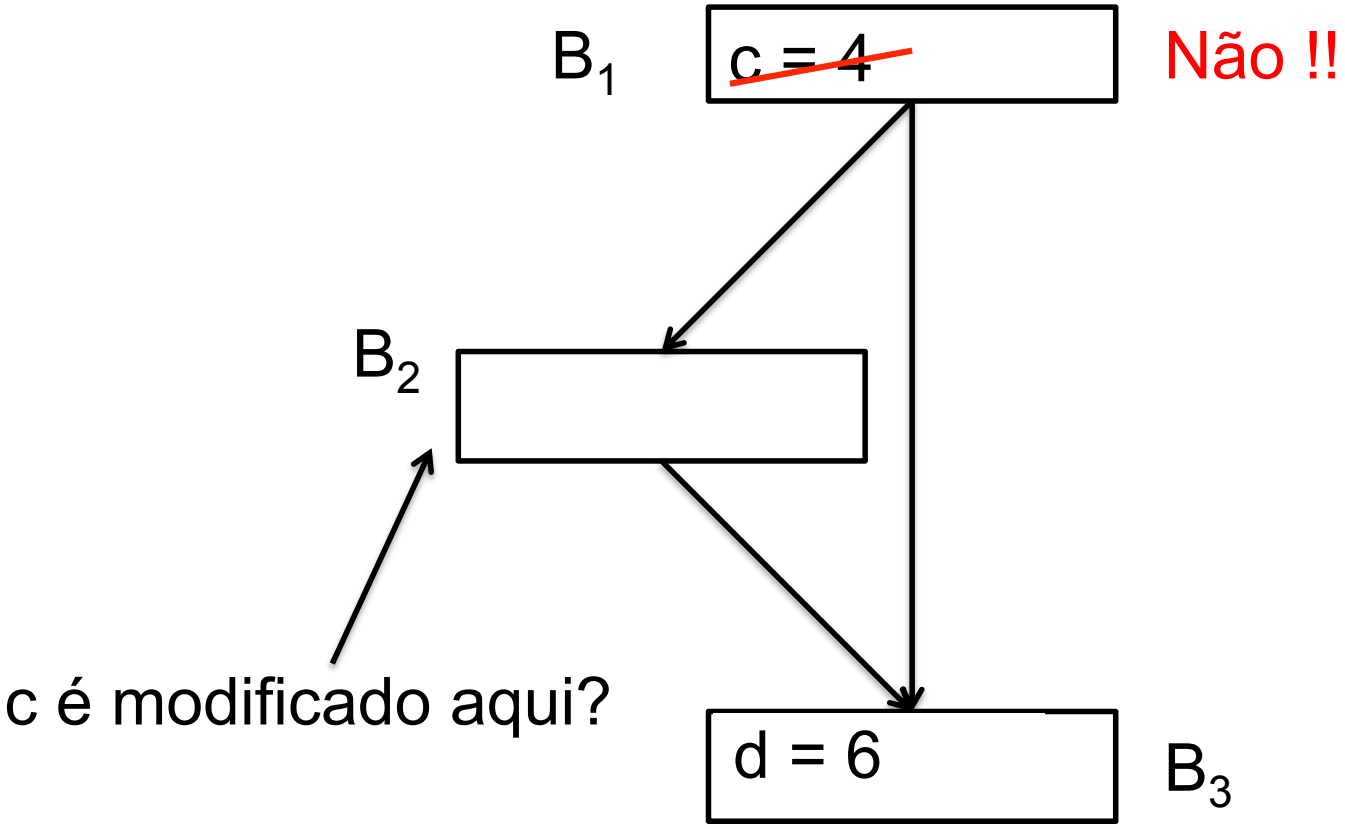
Principais Fontes de Otimização

- Transformações que preservam a funcionalidade
  - Eliminação de Sub-expressões comuns (CSE)
  - Propagação de Cópias
  - Eliminação de código morto
  - Constant folding
- Transformações Locais
  - Dentro de um bloco básico
- Transformações Globais
  - Envolve mais de um bloco básico
- Livro do dragão: seção 9.1

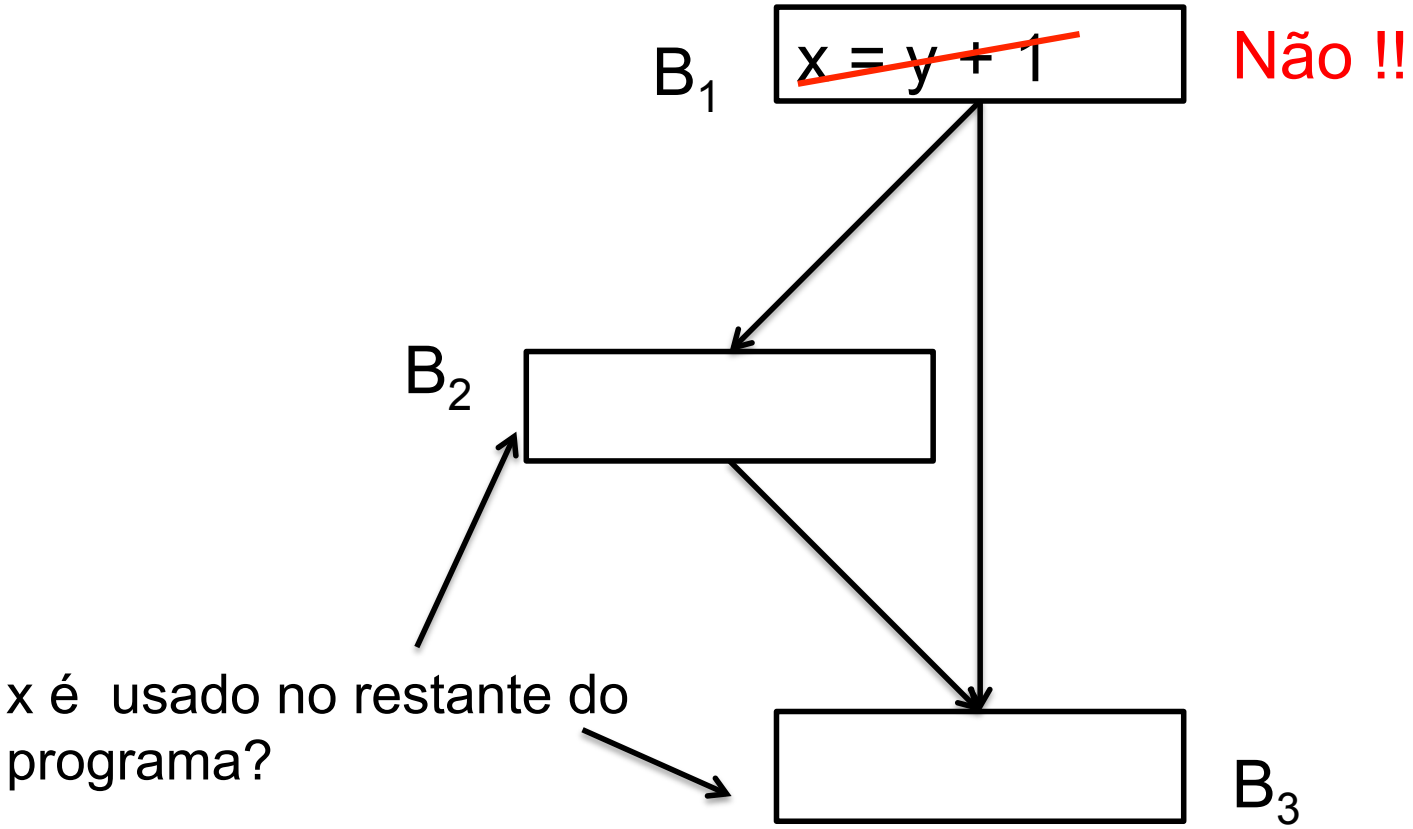
Copy Propagation



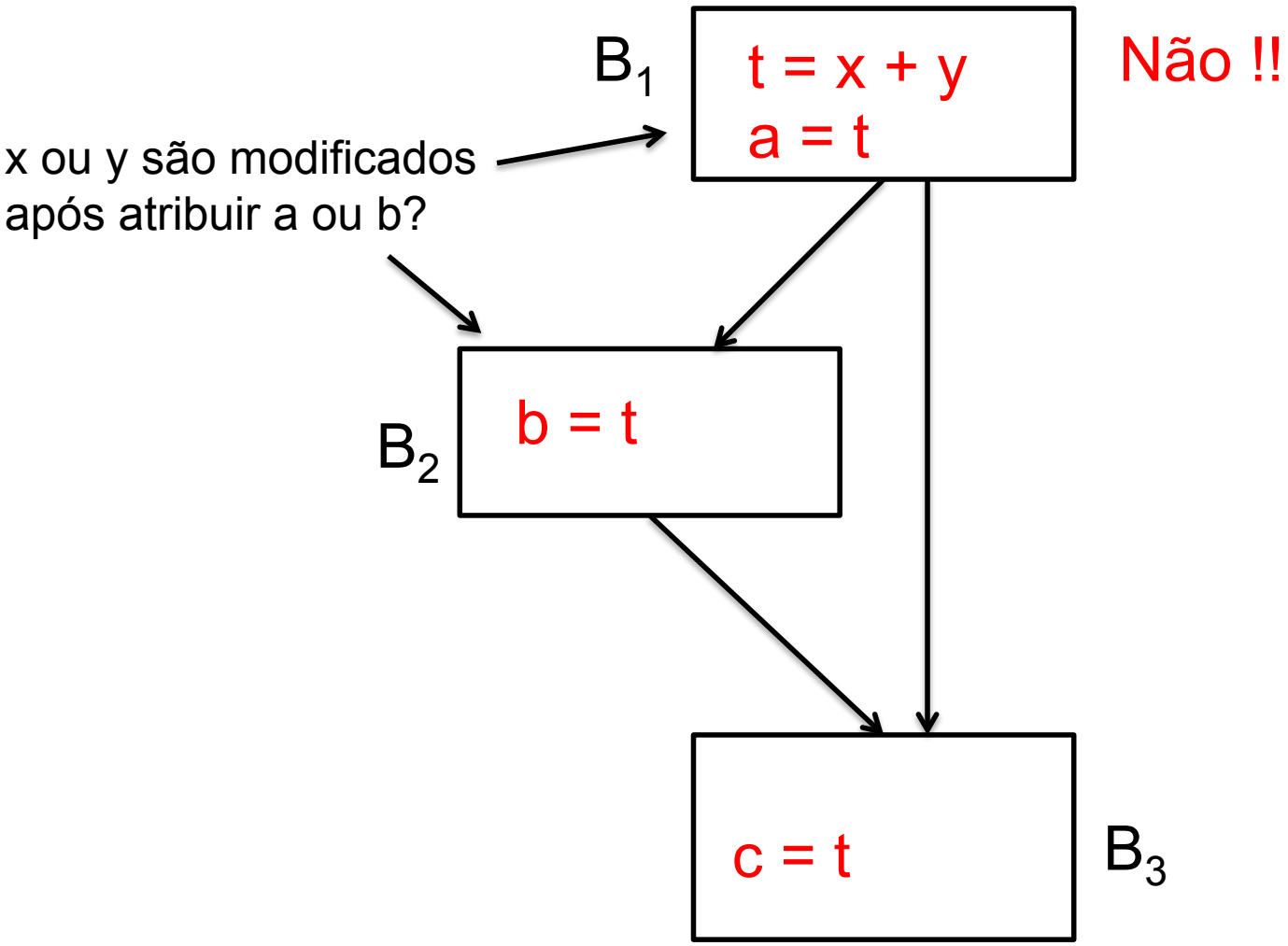
Constant Propagation



Dead Code Elimination



Common Sub-expression Elimination



Quick Sort

```
void quicksort(m,n)
int m,n;
{
     int i,j;
     int v,x;
     if ( n <= m ) return;
     /* fragment begins here */
     i = m-1; j = n; v = a[n];
     while(1) {
         do i = i+1; while ( a[i] < v );
         do j = j-1; while ( a[j] > v );
         if (i >= j ) break;
         x = a[i]; a[i] = a[j]; a[j] = x;
     }
     x = a[i]; a[i] = a[n]; a[n] = x;
     /* fragment ends here */
     quicksort(m,j); quicksort(i+1,n);
}
```

Fig. 10.2. C code for quicksort.

Quick Sort (original)

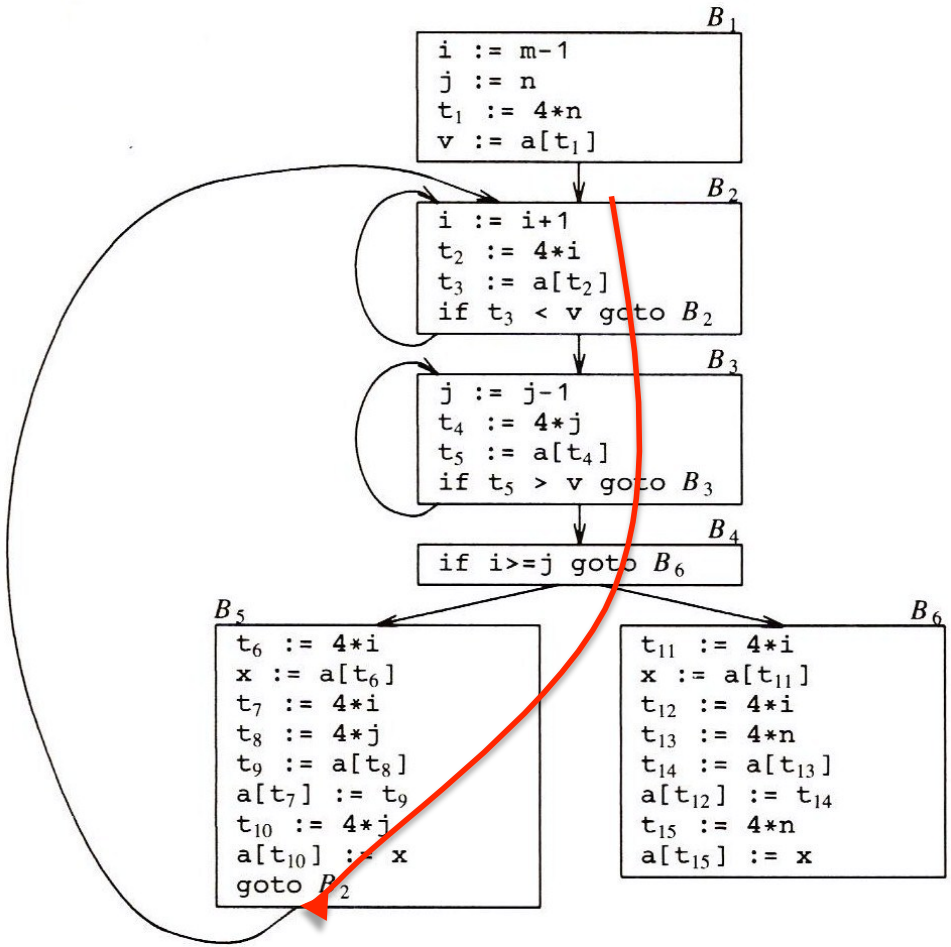


Fig. 10.5. Flow graph.

18 instruções

Quick Sort (otimizado)

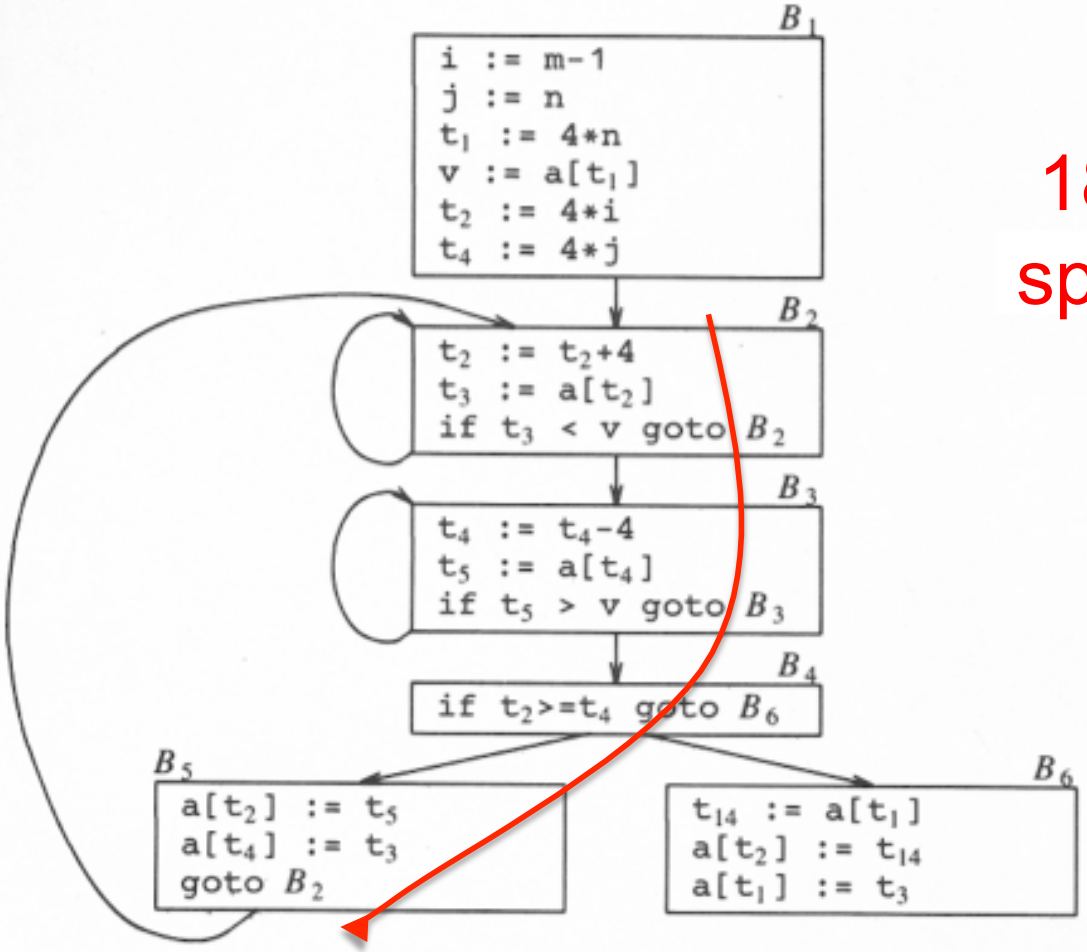


Fig. 10.10. Flow graph after induction-variable elimination.

18 instruções
speedup = 18/10
= 1.8

Análise de Fluxo de Dados

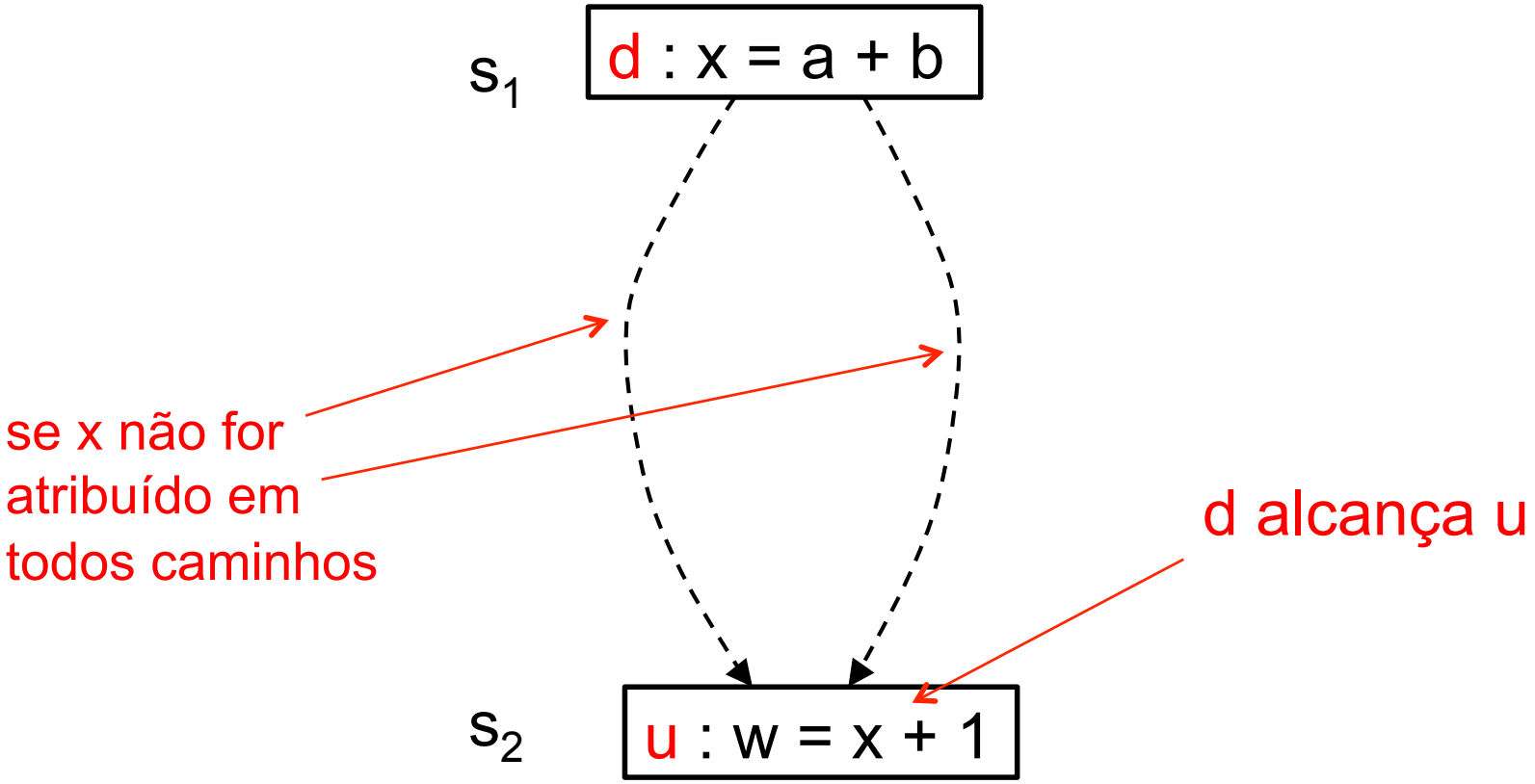
Introdução

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

Reaching Definitions

- Definição não ambígua de t:
  - d: t ← a op b
  - d: t ← M[a]
- d alcança uma sentença u:
  - 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
    - Não acontecem no compilador Minijava

Reaching Definitions



Reaching Definitions

- Pode ser expressa como equações de fluxo de dados
- 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): conjunto de todas as definições de t

Conjuntos Gen e Kill

Table 17.2: Gen and kill for reaching definitions.

Statement *s*	*gen*[s]	*kill*[s]
$d : t \leftarrow b \oplus c$	{*d*}	$defs(t) - \{d\}$
$d : t \leftarrow M[b]$	{*d*}	$defs(t) - \{d\}$
$M[a] \leftarrow b$	{}	{}
if *a* relop *b* goto L_1 else goto L_2	{}	{}
goto *L*	{}	{}
L :	{}	{}
$f(a_1, \ldots, a_n)$	{}	{}
$d : t \leftarrow f(a_1, \ldots, a_n)$	{*d*}	$defs(t) - \{d\}$

Reaching Definitions

- Usando gen e kill computamos:
  - In[n]: conjunto de definições que alcançam o início de n
  - Out[n]: conjunto de definições que alcançam o final de n

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

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

- In e Out inicializados com vazio.

Reaching Definitions

1 : a ← 5

2 : c ← 1

3 : L1 : if c > a goto L2

4 : c ← c + c

5 : goto L1

6 : L2 : a ← c - a

7 : c ← 0

Reaching Definitions

			Iter. 1	
n	*gen*[*n*]	*kill*[*n*]	*in*[*n*]	*out*[*n*]
1	1	6		1
2	2	4,7	1	1,2
3			1,2	1,2
4	4	2,7	1,2	1,4
5			1,4	1,4
6	6	1	1,2	2,6
7	7	2,4	2,6	6,7

1 : a ← 5

2 : c ← 1

3 : L1 : if c > a goto L2

4 : c ← c + c

5 : goto L1

6 : L2 : a ← c - a

7 : c ← 0

- Você imagina alguma otimização que poderia fazer no programa usando essa informação?

Reaching Definitions

			Iter. 1		Iter. 2		Iter. 3	
n	*gen*[*n*]	*kill*[*n*]	*in*[*n*]	*out*[*n*]	*in*[*n*]	*out*[*n*]	*in*[*n*]	*out*[*n*]
1	1	6		1		1		1
2	2	4,7	1	1,2	1	1,2	1	1,2
3			1,2	1,2	1,2,4	1,2,4	1,2,4	1,2,4
4	4	2,7	1,2	1,4	1,2,4	1,4	1,2,4	1,4
5			1,4	1,4	1,4	1,4	1,4	1,4
6	6	1	1,2	2,6	1,2,4	2,4,6	1,2,4	2,4,6
7	7	2,4	2,6	6,7	2,4,6	6,7	2,4,6	6,7

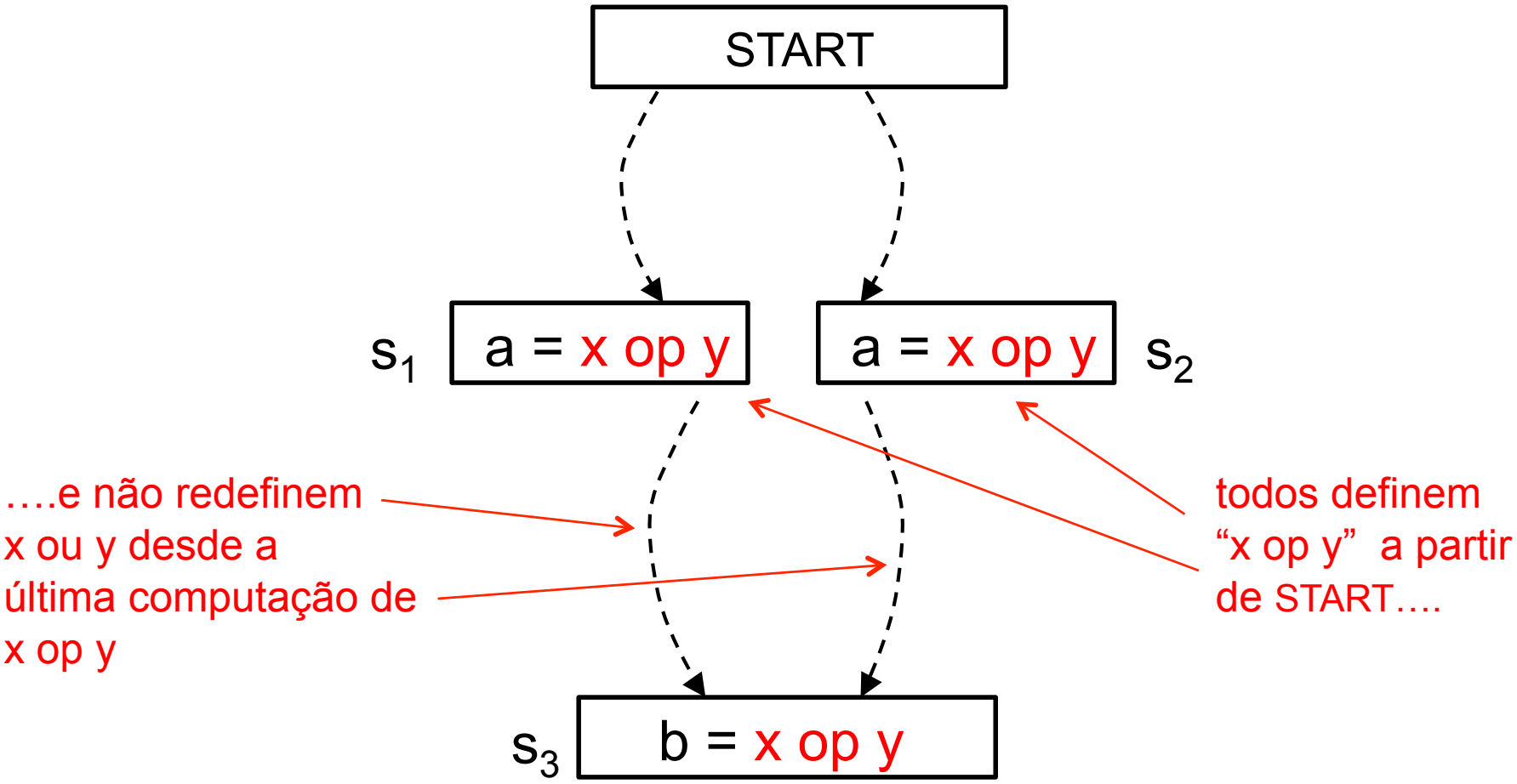
Reaching Definitions

$W \leftarrow$ the set of all nodes
while W is not empty
  remove a node n from W
  $old \leftarrow out[n]$
  $in \leftarrow \bigcup_{p \in pred[n]} out[p]$
  $out[n] \leftarrow gen[n] \cup (in - kill[n])$
  if $old \neq out[n]$
    for each successor s of n
      if $s \notin W$
        put s into W

Available Expressions

- *x op y* está disponível em n no CFG se:
  - Para todo caminho a partir do nó de entrada até n, *x op y* é computada pelo menos uma vez
  - Não há definições de x ou y após a mais recente ocorrência de *x op y* no caminho
- Gen e kill se tornam conjuntos de expressões
  - Nó que calcula x op y: Gera *x op y*
  - Qualquer definição de x ou y mata *x op*

Available Expressions



Available Expressions

Table 17.4: *Gen* and *kill* for available expressions.

Statement *s*	*gen*[*s*]	*kill*[*s*]
$t \leftarrow b \oplus c$	$\{b \oplus c\} - kill[s]$	*expressions containing t*
$t \leftarrow M[b]$	$\{M[b]\} - kill[s]$	*expressions containing t*
$M[a] \leftarrow b$	{}	*expressions of the form M[x]*
if $a > b$ goto L_1 else goto L_2	{}	{}
goto L	{}	{}
L :	{}	{}
$f(a_1, \ldots, a_n)$	{}	*expressions of the form M[x]*
$t \leftarrow f(a_1, \ldots, a_n)$	{}	*expressions containing t, and expressions of the form M[x]*

Available Expressions

- Usando gen e kill computamos:
  - In[n]: conjunto de expressões disponíveis no início de n
  - Out[n]: conjunto de expressões disponíveis no final de n

$$in[n] = \bigcap_{p \in pred[n]} out[p] \qquad \text{if } n \text{ is not the start node}$$

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

- In e Out inicializados com “cheio”.
  - Por que?
- Exceção para in do nó de entrada

Liveness Analysis

- Podemos usar gen e kill:
  - Usos de variável geram liveness
  - Definições de variável matam liveness

Statement *s*	*gen*[*s*]	*kill*[s]
$t \leftarrow b \oplus c$	$\{b, c\}$	$\{t\}$
$t \leftarrow M[b]$	$\{b\}$	$\{t\}$
$M[a] \leftarrow b$	$\{a, b\}$	$\{\}$
if $a > b$ goto L_1 else goto L_2	$\{a, b\}$	$\{\}$
goto L	$\{\}$	$\{\}$
L :	$\{\}$	$\{\}$
$f(a_1, \ldots, a_n)$	$\{a_1, \ldots, a_n\}$	$\{\}$
$t \leftarrow f(a_1, \ldots, a_n)$	$\{a_1, \ldots, a_n\}$	$\{t\}$

Liveness Analysis

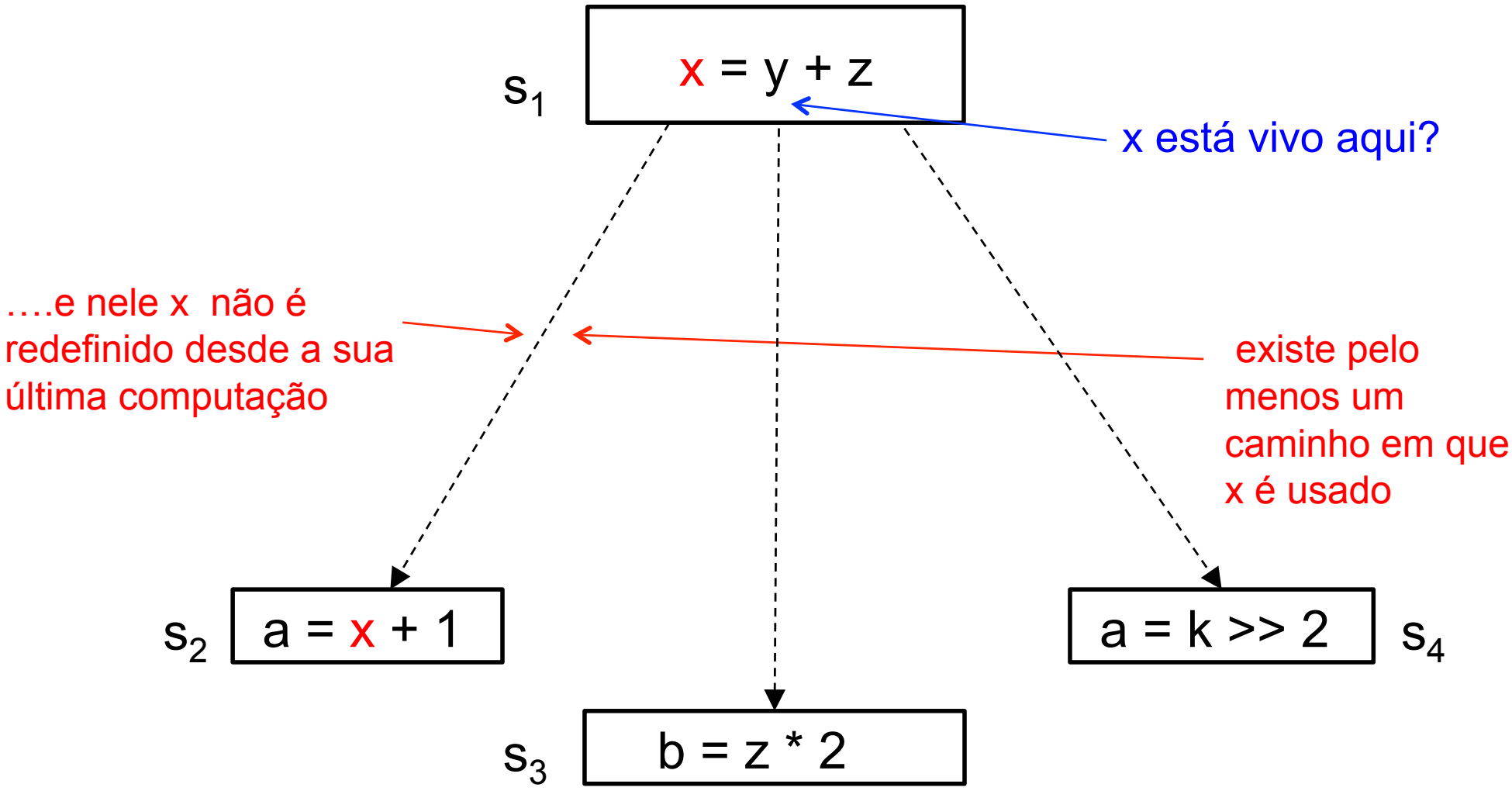
- Podemos usar gen e kill:
  - Usos de variável geram liveness
  - Definições de variável matam liveness

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

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

- Liveness tem fluxo contrário ao do grafo

Liveness Analysis



Blocos Básicos

- Suponha dois nós no CFG n e p
  - p é o único predecessor de n
- Neste caso, podemos combinar os efeitos gen e kill de n e p
- Teremos apenas um nó no grafo
- Podemos repetir para todas as instruções de um bloco básico!

Blocos Básicos

- Bloco básico: Apenas uma entrada, uma saída e nenhum desvio contido nele.
- Pense em Reaching Definitions
  - Como combinar gen e kill para um bloco básico?
- *out*[*n*] = *gen*[*n*] U (*in*[*n*] − *kill*[*n*]).
- in[n] = out[p]. Por quê?

Blocos Básicos

- Então temos:
  - $out[n] = gen[n] \cup ((gen[p] \cup (in[p] - kill[p])) - kill[n])$.
- Lembre-se que
  - $(A \cup B) - C = (A - C) \cup (B - C)$
  - $A - (B \cup C) = (A - B) - C$
- Logo:
  - $out[n] = gen[n] \cup ((gen[p] - kill[n]) \cup (in[p] - (kill[p] \cup kill[n])))$

Blocos Básicos

- Logo:
  - $out[n] = gen[n] \cup ((gen[p] - kill[n]) \cup (in[p] - (kill[p] \cup kill[n]))$
- Daí tiramos que:
  - $gen[pn] = gen[n] \cup (gen[p] - kill[n])$
  - $kill[pn] = kill[p] \cup kill[n]$
- Exercício: Deduza essas equações para outras análises
  - Available expressions, liveness

Blocos Básicos

- Usando essa técnica podemos
  - Combinar todas as sentenças de um bloco básico
  - Criar gen e kill para o bloco todo
- O CFG de BBs é muito menor que o de sentenças individuais
- Acelera a análise

Ordenação dos Nós

- Forward analysis:
  - Ordenar os nós com DFS
  - Topológica (sem ciclos)
  - Quase-topológica (com ciclos)
  - Faz com que a maioria dos predecessores seja computada antes dos sucessores
- Backward analysis
  - Começar pelo nó saída