INTRODUCTION À L'ANALYSE DE LA STRUCTURE DES PROGRAMMES

Librement adapté du chapitre 6 de Software Testing and Analysis de Pezzè et Young

VV034
v201a

2014-10-01

Luc LAVOIE
Département d'informatique
Faculté des sciences

UNIVERSITÉ DE SHERBROOKE

Luc.Lavoie@USherbrooke.ca
http://info.usherbrooke.ca/llavoie

POURQUOI ANALYSER LA STRUCTURE DES PROGRAMMES ?

- Détecter automatiquement et systématiquement
  - des anomalies
    - voire des erreurs et mêmes des défauts!
  - des incohérences
    - grâce au principe de redondance
  - ...

POURQUOI UTILISER UN MODÈLE DE GRAPHE ET DES ALGORITHMES DE FLUX ?

- Le graphe représente bien l'aspect *statique* du programme
- Les flux représente bien l'aspect *dynamique* du programme, tant du point de vue
  - des données (*data*)
  - de la commande (*control*)
- Plusieurs questions naturelles se posent facilement en ces termes
  - D'où provient la valeur d'une variable?
  - Quel est l'effet d'un changement d'une instruction précise?
  - ...
- Plusieurs problèmes trouvent une solution (partielle) grâce aux algorithmes d'analyse de flux
  - Allocation/libération dynamique de mémoire
  - Injection de code SQL

CIBLES D'APPRENTISSAGE

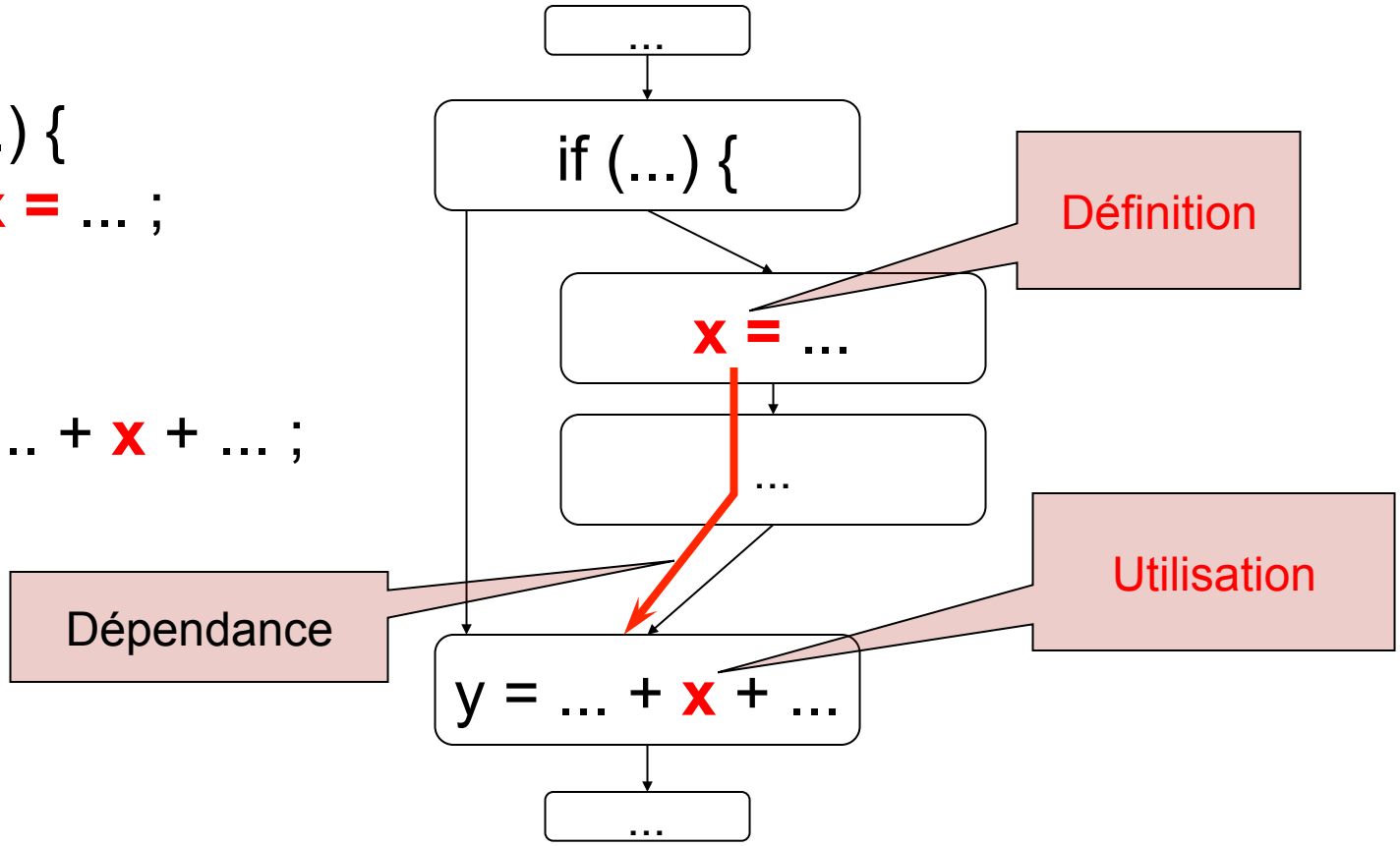
- Comprendre les base de la modélisation de programme à l'aide de graphe et de flux :
  - dépendance, dominance, etc. *(def-use pairs, dominators…)*
- Connaitre les possibilités offertes par certains algorithmes fondamentaux et les techniques qui en découlent
  - Construction de modèles adaptés à la solution de problèmes particuliers
  - Évaluation et analyse de propriétés calculées
- Connaitre et comprendre les limites pratiques de l'utilisation de ces techniques

DÉPENDANCE (1)

- Dans un programme, une dépendance (*def-use pair*) associe une instruction calculant une valeur à une autre l'utilisant.
- Le plus souvent, il s'agit d'une définition de variable vers son utilisation.
- Définition de variable
  - Déclaration
    - avec initialisation explicite
    - avec initialisation implicite (éventuellement « non initialisée »)
  - Initialisation d'un paramètre local
  - Affectation
- Utilisation de variable
  - Expression
    - affectation, conditionnelle, boucle...
    - paramètre effectif
    - retour de fonction

DÉPENDANCE (2)

```
...
if (...) {
    x = ... ;
...
}
y = ... + x + ... ;
```



DÉPENDANCE (3)

```
/**  Algorithme d'Euclide */
public class GCD
{
public int pgcd(int x, int y) {
    int tmp;              // A: def x, y, tmp
    while (y != 0) {     // B: use y
       tmp = x % y;     // C: def tmp; use x, y
       x = y;           // D: def x; use y
       y = tmp;         // E: def y; use tmp
    }
    return x;           // F: use x
  }
```

def : définition
use : utilisation

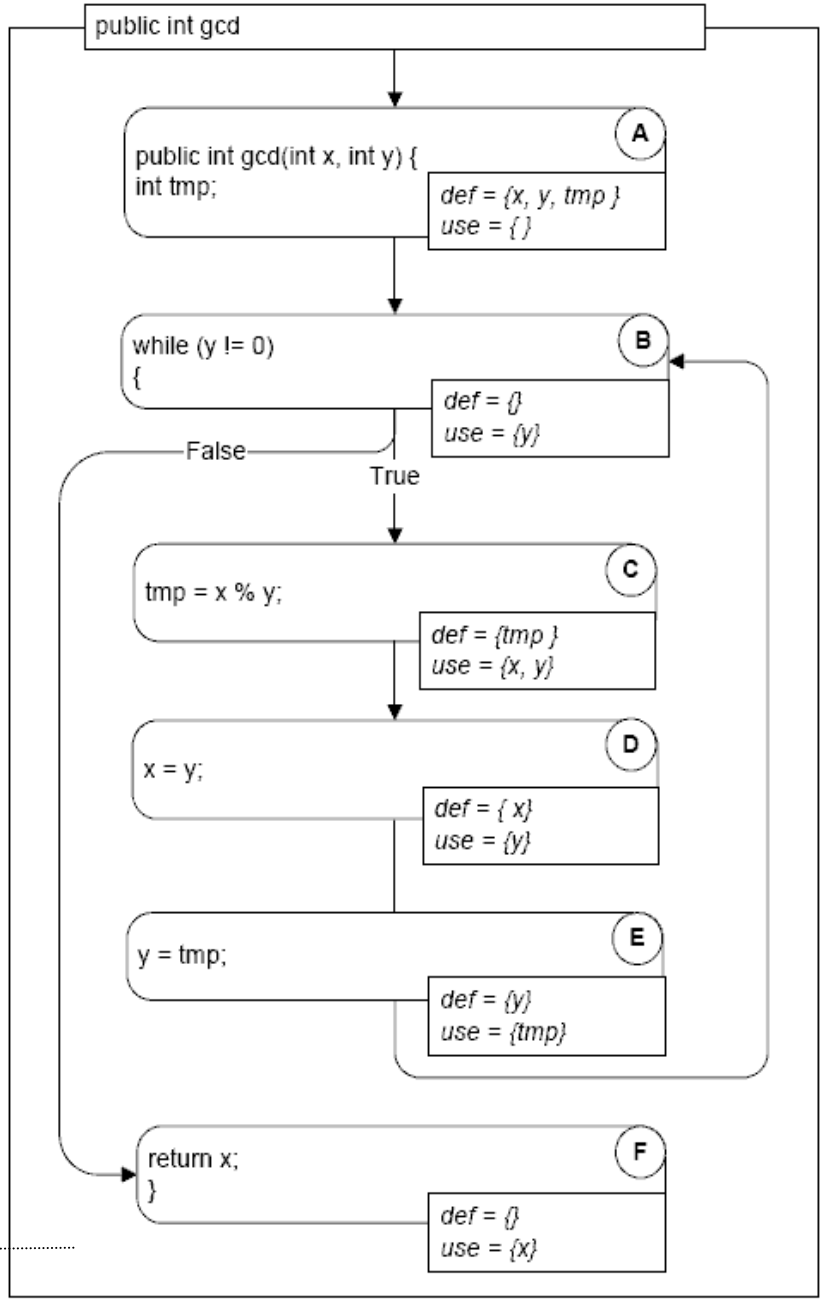


Figure 6.2, page 79

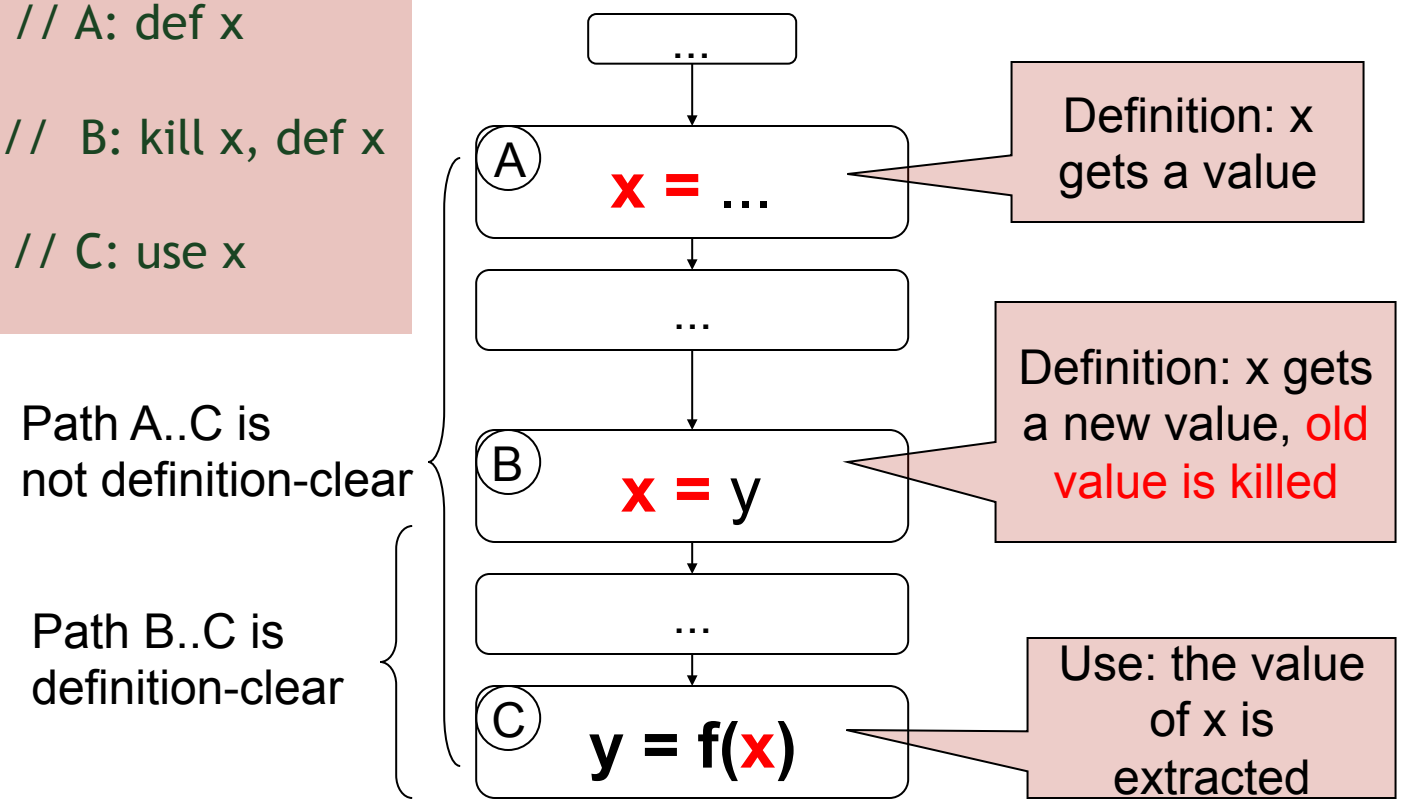
DÉPENDANCE (4)

- Un **chemin exempt de redéfinition** (*definition-clear path*) est un chemin (du graphe) du programme réunissant la définition d'une variable v à une utilisation de v ne passant pas(*) par une (re)définition de v.
  - S'il existe une telle redéfinition, on dit qu'elle oblitère (*kills*) la précédente.
- Un point de définition et un point d'utilisation forment une dépendance ssi il existe un chemin exempt de redéfinition entre eux.

**There is an over-simplification here, which we will repair later.*

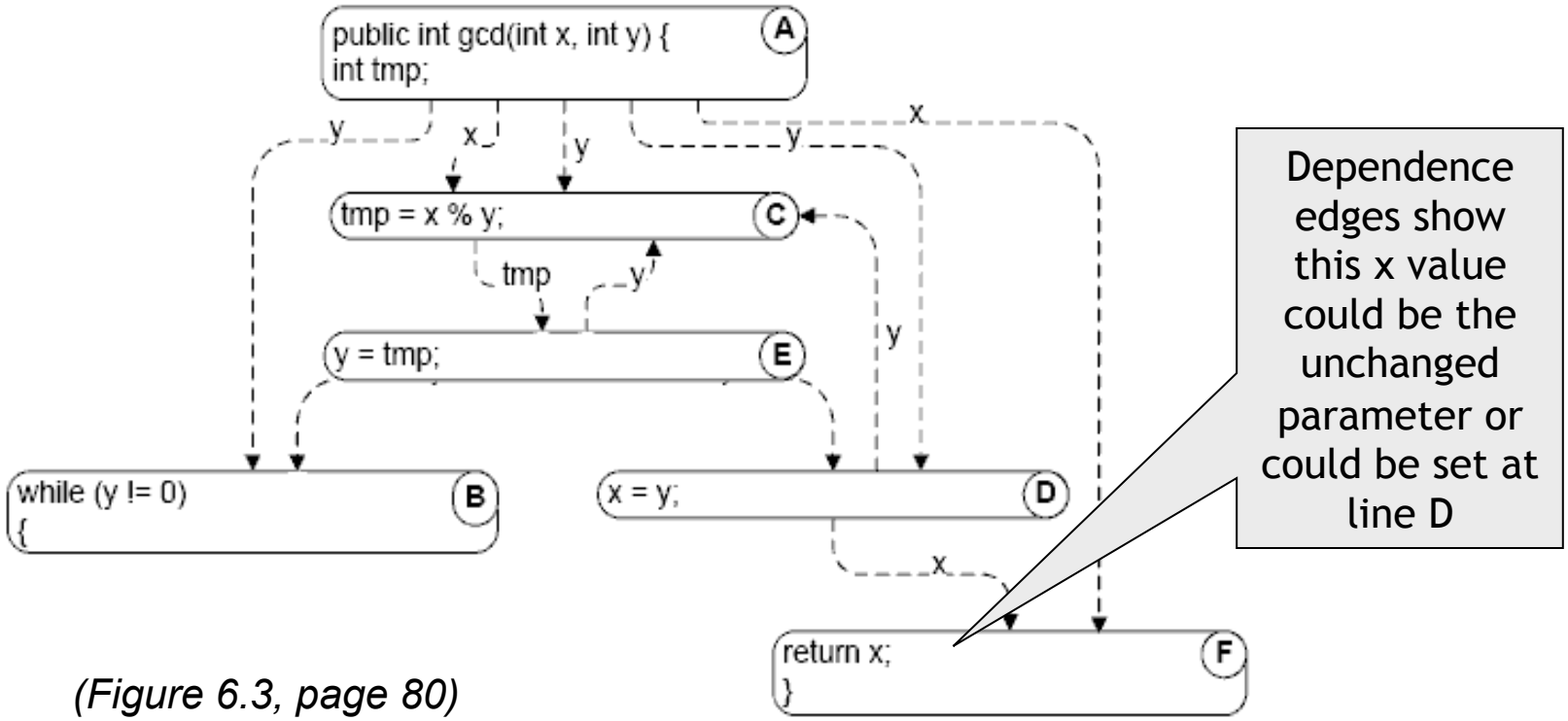
Definition-Clear or Killing

```
x = ...     // A: def x
q = ...
x = y;      //  B: kill x, def x
z = ...
y = f(x);   // C: use x
```



(DIRECT) DATA DEPENDENCE GRAPH

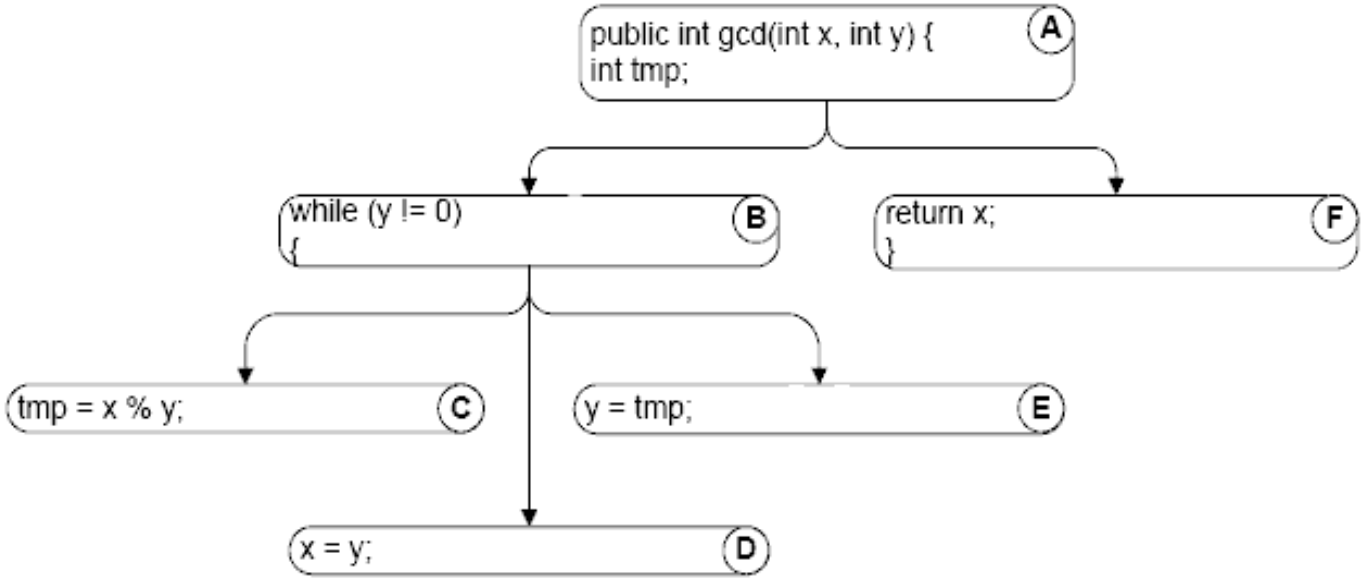
- A direct data dependence graph is:
  - Nodes: as in the control flow graph (CFG)
  - Edges: def-use (du) pairs, labelled with the variable name



(Figure 6.3, page 80)

CONTROL DEPENDENCE (1)

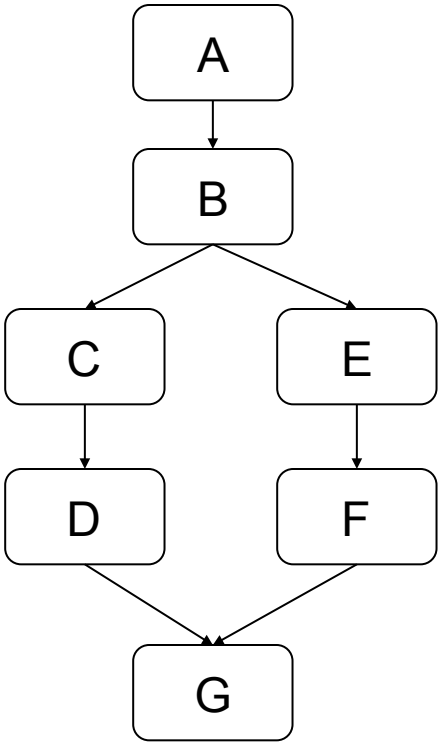
- Data dependence: Where did these values come from?
- Control dependence: Which statement controls whether this statement executes?
  - Nodes: as in the CFG
  - Edges: unlabelled, from entry/branching points to controlled blocks



DOMINATORS

- **Pre-dominators** in a rooted, directed graph can be used to make this intuitive notion of "controlling decision" precise.
- Node M dominates node N if every path from the root to N passes through M.
  - A node will typically have many dominators, but except for the root, there is a unique **immediate dominator** of node N which is closest to N on any path from the root, and which is in turn dominated by all the other dominators of N.
  - Because each node (except the root) has a unique immediate dominator, the immediate dominator relation forms a tree.
- **Post-dominators**: Calculated in the reverse of the control flow graph, using a special "exit" node as the root.

DOMINATORS (EXAMPLE)

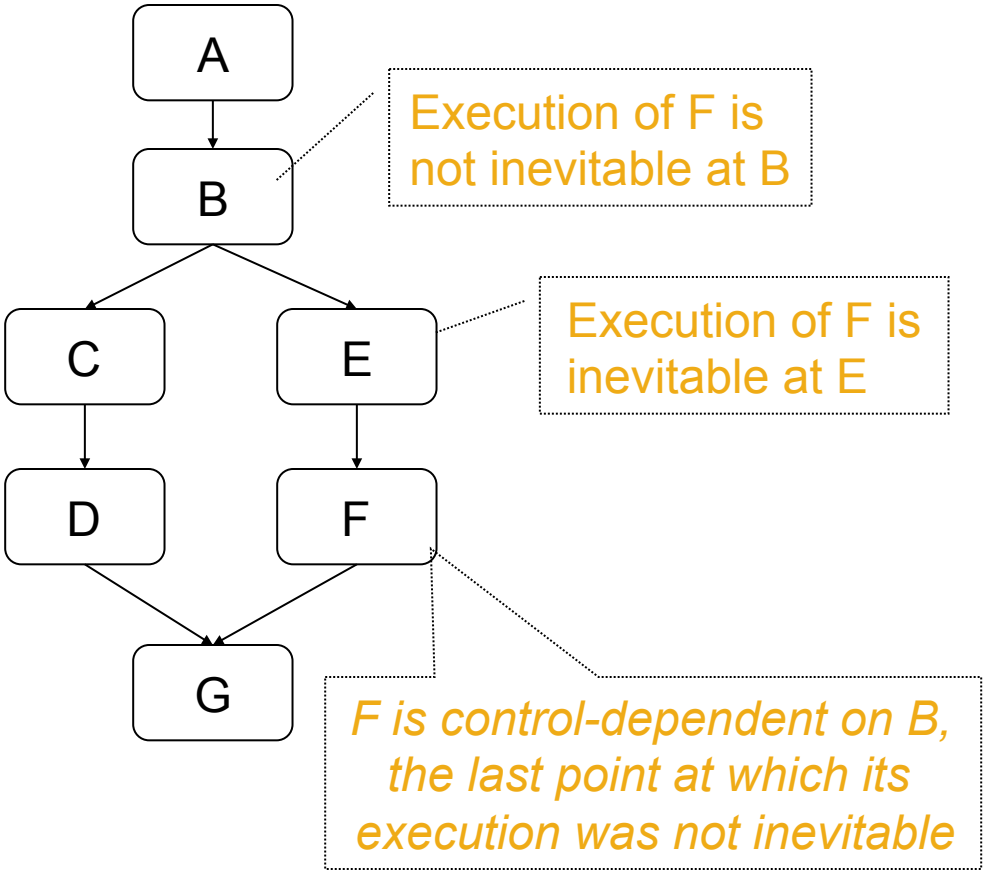


- A pre-dominates all nodes; G post-dominates all nodes
- F and G post-dominate E
- G is the immediate post-dominator of B
  - C does *not* post-dominate B
- B is the immediate pre-dominator of G
  - F does *not* pre-dominate G

CONTROL DEPENDENCE (2)

- We can use post-dominators to give a more precise definition of control dependence:
  - Consider again a node N that is reached on some but not all execution paths.
  - There must be some node C with the following property:
    - C has at least two successors in the control flow graph (i.e., it represents a control flow decision);
    - C is not post-dominated by N
    - there is a successor of C in the control flow graph that is post-dominated by N.
  - When these conditions are true, we say node N is control-dependent on node C.
    - Intuitively: C was the last decision that controlled whether N executed

CONTROL DEPENDENCE



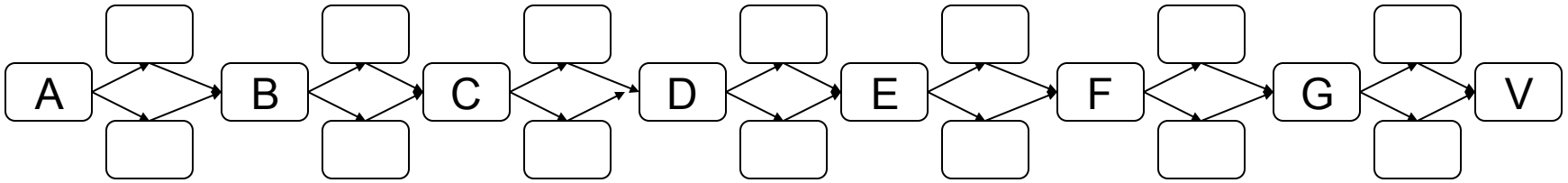
Data Flow Analysis

- Computing data flow information

CALCULATING DEF-USE PAIRS

- Definition-use pairs can be defined in terms of paths in the program control flow graph:
  - There is an association (d,u) between a definition of variable v at d and a use of variable v at u iff
    - there is at least one control flow path from d to u
    - with no intervening definition of v.
  - v_d **reaches** u (v_d is a **reaching definition** at u).
  - If a control flow path passes through another definition e of the same variable v, v_e **kills** v_d at that point.
- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges.
- Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

EXPONENTIAL PATHS (EVEN WITHOUT LOOPS)



2 paths from A to B

4 from A to C

8 from A to D

16 from A to E

...

128 paths from A to V

Tracing each path is not efficient, and we can do much better.

DF ALGORITHM

- An efficient algorithm for computing reaching definitions (and several other properties) is based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node.
- Suppose we are calculating the reaching definitions of node n, and there is an edge (p,n) from an immediate predecessor node p.
  - If the predecessor node p can assign a value to variable v, then the definition v_p reaches n. We say the definition v_p is generated at p.
  - If a definition v_p of variable v reaches a predecessor node p, and if v is not redefined at that node (in which case we say the v_p is killed at that point), then the definition is propagated on from p to n.

EQUATIONS OF NODE E (Y = TMP)

Calculate reaching definitions at E in terms of its immediate predecessor D

```
public class GCD  {
public int gcd(int x, int y) {
    int tmp;              // A: def x, y, tmp
    while (y != 0) {      // B: use y
       tmp = x % y;       // C: def tmp; use x, y
       x = y;             // D: def x; use y
       y = tmp;           // E: def y; use tmp
    }
    return x;             // F: use x
   }
```

$$\text{Reach}(E) = \text{ReachOut}(D)$$

$$\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$$

EQUATIONS OF NODE B (WHILE (Y != 0))

This line has two predecessors: Before the loop, end of the loop

```
public class GCD  {
public int gcd(int x, int y) {
    int tmp;              // A: def x, y, tmp
    while (y != 0) {      // B: use y
        tmp = x % y;      // C: def tmp; use x, y
        x = y;            // D: def x; use y
        y = tmp;          // E: def y; use tmp
    }
    return x;             // F: use x
   }
```

- $Reach(B) = ReachOut(A) \cup ReachOut(E)$
- $ReachOut(A) = gen(A) = \{x_A, y_A, tmp_A\}$
- $ReachOut(E) = (Reach(E) \setminus \{y_A\}) \cup \{y_E\}$

General equations for Reach analysis

$$\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$$

$$\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v_n \mid v \text{ is defined or modified at } n \}$$

$$\text{kill}(n) = \{ v_x \mid v \text{ is defined or modified at } x, x \neq n \}$$

AVAIL EQUATIONS

$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ \text{exp} \mid \text{exp is computed at n} \}$$

$$\text{kill}(n) = \{ \text{exp} \mid \text{exp has variables assigned at n} \}$$

LIVE VARIABLE EQUATIONS

$$\text{Live}(n) = \bigcup_{m \in \text{succ}(n)} \text{LiveOut}(m)$$

$$\text{LiveOut}(n) = (\text{Live}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v \mid v \text{ is used at } n \}$$

$$\text{kill}(n) = \{ v \mid v \text{ is modified at } n \}$$

CLASSIFICATION OF ANALYSES

- Forward/backward: a node's set depends on that of its predecessors/successors
- Any-path/all-path: a node's set contains a value iff it is coming from any/all of its inputs

	Any-path (∪)	All-paths (∩)
Forward (pred)	Reach	Avail
Backward (succ)	Live	"inevitable"

Iterative Solution of Dataflow Equations

- Initialize values (first estimate of answer)
  - For "any path" problems, first guess is "nothing" (empty set) at each node
  - For "all paths" problems, first guess is "everything" (set of all possible values = union of all "gen" sets)
- Repeat until nothing changes
  - Pick some node and recalculate (new estimate)

This will converge on a "fixed point" solution where every new calculation produces the same value as the previous guess.

WORKLIST ALGORITHM FOR DATA FLOW

See figures 6.6, 6.7 on pages 84, 86 of Pezzè & Young

One way to iterate to a fixed point solution.

General idea:

- Initially all nodes are on the work list, and have default values
  - Default for "any-path" problem is the empty set, default for "all-path" problem is the set of all possibilities (union of all gen sets)
- While the work list is not empty
  - Pick any node n on work list; remove it from the list
  - Apply the data flow equations for that node to get new values
  - If the new value is changed (from the old value at that node), then
    - Add successors (for forward analysis) or predecessors (for backward analysis) on the work list
- Eventually the work list will be empty (because new computed values = old values for each node) and the algorithm stops.

UN CLASSIQUE AVEC REACH, LIVE AND AVAIL

```
static void questionable () {
   int k;
   for (int i = 0; i < 10; ++i) {
      if (cond(i)) {
         k=0;
         ...
      } else {
         k+=i;
         ...
      }
   proc (k);
   }
}
```

- Version statique
- Version dynamique
- Version couverture

Cooking your own: From Execution to Conservative Flow Analysis

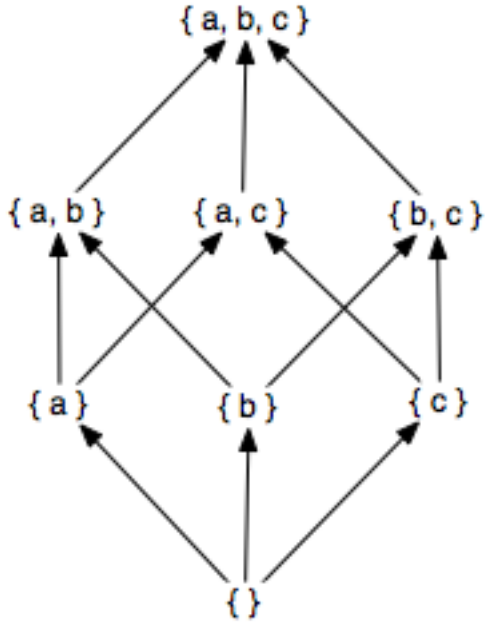
- We can use the same data flow algorithms to approximate other dynamic properties
  - Gen set will be "facts that become true here"
  - Kill set will be "facts that are no longer true here"
  - Flow equations will describe propagation
- Example: Taintedness (in web form processing)
  - "Taint": a user-supplied value (e.g., from web form) that has not been validated
  - Gen: we get this value from an untrusted source here
  - Kill: we validated to make sure the value is proper

COOKING YOUR OWN ANALYSIS (2)

- Flow equations must be monotonic
  - Initialize to the bottom element of a lattice of approximations
  - Each new value that changes must move up the lattice
- Typically: Powerset lattice
  - Bottom is empty set, top is universe
  - Or empty at top for all-paths analysis

Monotonic: y > x implies f(y) ≥ f(x)

(where f is application of the flow equations on values from successor or predecessor nodes, and “>” is movement up the lattice)



DATA FLOW ANALYSIS WITH ARRAYS AND POINTERS

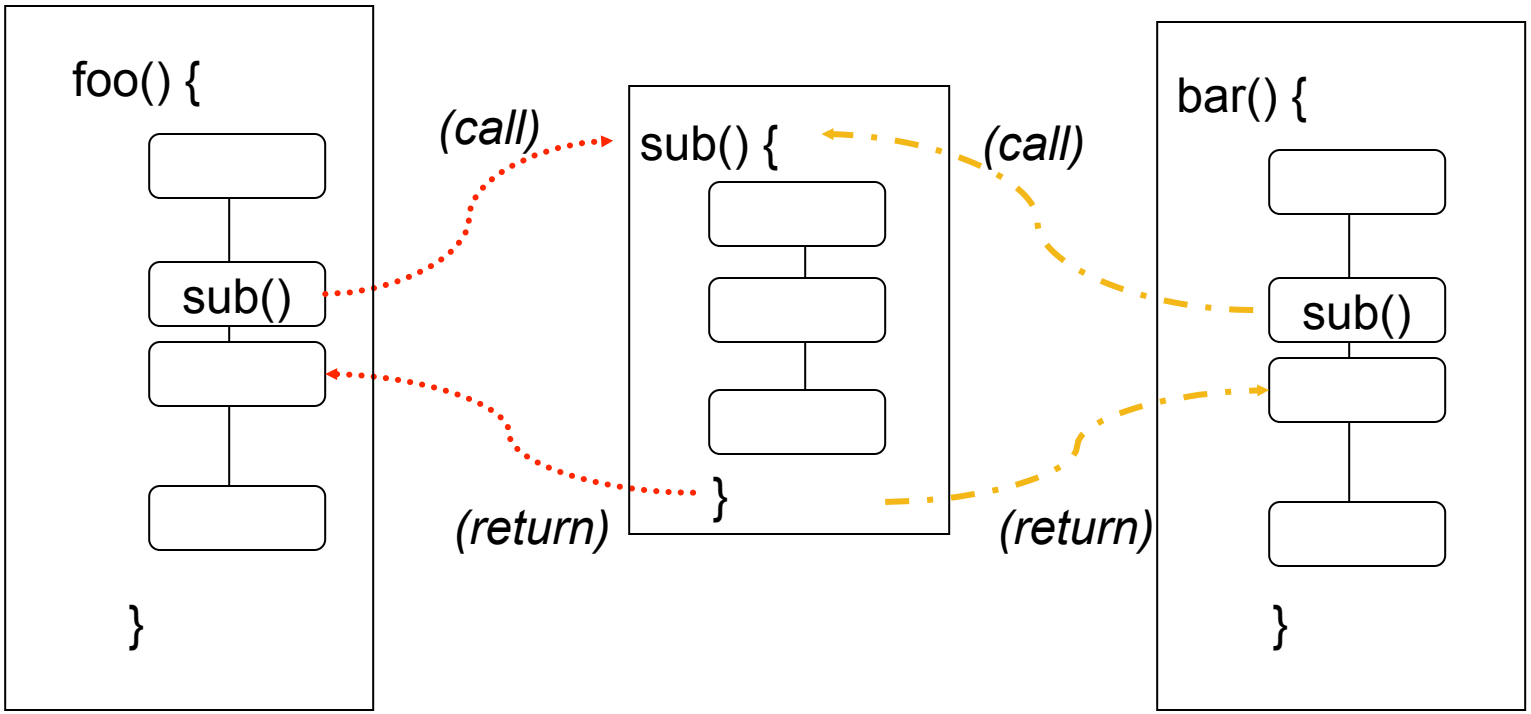
- Arrays and pointers introduce uncertainty:
  Do different expressions access the same storage?
  - a[i] same as a[k] when i = k
  - a[i] same as b[i] when a = b (**aliasing**)
- The uncertainty is accomodated depending to the kind of analysis
  - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
  - All-path: vice versa

Scope of Data Flow Analysis

- Intraprocedural
  - Within a single method or procedure
    - as described so far
- Interprocedural
  - Across several methods (and classes) or procedures

- Cost/Precision trade-offs for interprocedural analysis are critical, and difficult
  - context sensitivity
  - flow-sensitivity

CONTEXT SENSITIVITY



A **context-sensitive** (interprocedural) analysis distinguishes sub() called from foo() from sub() called from bar();
A **context-insensitive** (interprocedural) analysis does not separate them, as if foo() could call sub() and sub() could then return to bar()

Flow Sensitivity

- Reach, Avail, etc. were flow-sensitive, intraprocedural analyses
  - They considered ordering and control flow decisions
  - Within a single procedure or method, this is (fairly) cheap — $O(n^3)$ for n CFG nodes
- Many interprocedural flow analyses are flow-insensitive
  - $O(n^3)$ would not be acceptable for all the statements in a program!
    - Though $O(n^3)$ on each individual procedure might be ok
  - Often flow-insensitive analysis is good enough ... consider type checking as an example

SUMMARY

- Data flow models detect patterns on CFGs:
  - Nodes initiating the pattern
  - Nodes terminating it
  - Nodes that may interrupt it
- Often, but not always, about flow of information (dependence)
- Pros:
  - Can be implemented by efficient iterative algorithms
  - Widely applicable (not just for classic “data flow” properties)
- Limitations:
  - Unable to distinguish feasible from infeasible paths
  - Analyses spanning whole programs (e.g., alias analysis) must trade off precision against computational cost