Data-Flow Analysis

Chapter 9, Section 9.2, 9.3, 9.4

Data-Flow Analysis

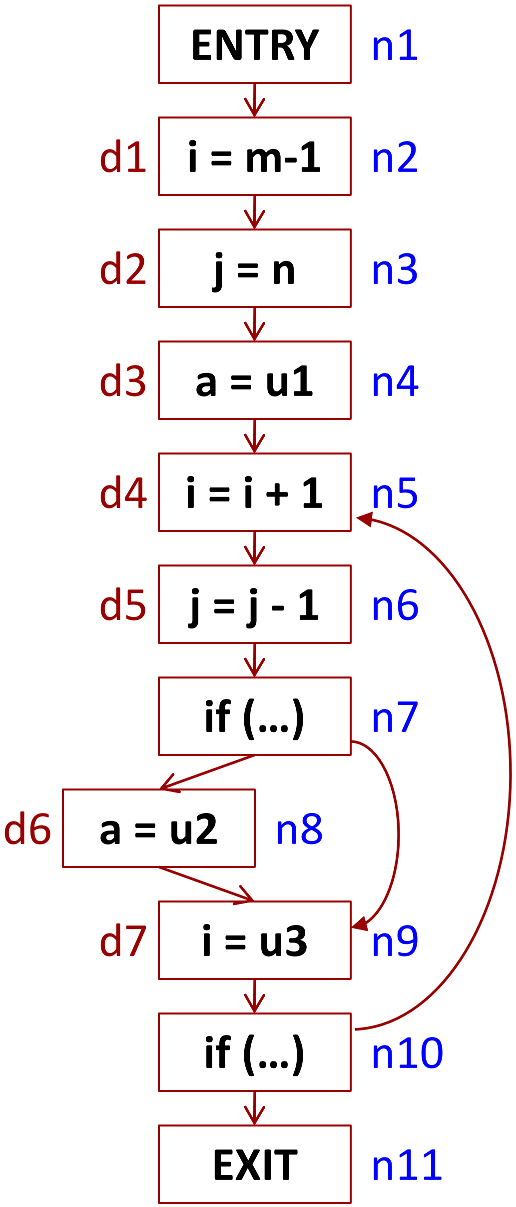
- Data-flow analysis is a sub-area of static program analysis (aka compile-time analysis)
  - Used in the compiler back end for optimizations of three-address code and for generation of target code
  - For software engineering tools: software understanding, restructuring, testing, verification
- Attaches to each CFG node some information that describes properties of the program at that point
  - Based on lattice theory
- Defines algorithms for inferring these properties
  - e.g., fixed-point computation

Example: Reaching Definitions

- A classical example of a data-flow analysis
  - We will consider **intra**procedural analysis: only inside a single procedure, based on its CFG
- For ease of discussion, pretend that the CFG nodes are individual instructions, not basic blocks
  - Each node defines two program points: immediately before and immediately after
- Goal: identify all connections between variable definitions ("write") and variable uses ("read")
  - **x = y + z** has a definition of **x** and uses of **y** and **z**

Reaching Definitions

- A definition *d* reaches a program point *p* if there exists a CFG path that
  - starts at the program point immediately after *d*
  - ends at *p*
  - does **not** contain a definition of *d* (i.e., *d* is not "killed")
- The CFG path may be impossible (*infeasible*) at run time
  - Any compile-time analysis has to be *conservative*, so we consider all paths in the CFG
- For a CFG node *n*
  - IN[*n*] is the set of definitions that reach the program point immediately before *n*
  - OUT[*n*] is the set of definitions that reach the program point immediately after *n*
  - Reaching definitions analysis computes IN[*n*] and OUT[*n*]



OUT[n1] = { }
IN[n2] = { }
OUT[n2] = { d1 }
IN[n3] = { d1 }
OUT[n3] = { d1, d2 }
IN[n4] = { d1, d2 }
OUT[n4] = { d1, d2, d3 }
IN[n5] = { d1, d2, d3, d5, d6, d7 }
OUT[n5] = { d2, d3, d4, d5, d6 }
IN[n6] = { d2, d3, d4, d5, d6 }
OUT[n6] = { d3, d4, d5, d6 }
IN[n7] = { d3, d4, d5, d6 }
OUT[n7] = { d3, d4, d5, d6 }
IN[n8] = { d3, d4, d5, d6 }
OUT[n8] = { d4, d5, d6 }
IN[n9] = { d3, d4, d5, d6 }
OUT[n9] = { d3, d5, d6, d7 }
IN[n10] = { d3, d5, d6, d7 }
OUT[n10] = { d3, d5, d6, d7 }
IN[n11] = { d3, d5, d6, d7 }

Examples of relationships:
IN[n2] = OUT[n1]
IN[n5] = OUT[n4] ∪ OUT[n10]
OUT[n7] = IN[n7]
OUT[n9] = (IN[n9] – {d1,d4,d7}) ∪ {d7}

Formulation as a System of Equations

- For each CFG node *n*

$$\mathrm{IN}[n] = \bigcup_{m \in \mathrm{Predecessors}(n)} \mathrm{OUT}[m]$$

$$\mathrm{OUT}[\mathrm{ENTRY}] = \varnothing$$

$$\mathrm{OUT}[n] = (\mathrm{IN}[n] - \mathrm{KILL}[n]) \cup \mathrm{GEN}[n]$$

  – GEN[*n*] is a singleton set containing the definition *d* at *n*
  – KILL[*n*] is the set of all defs of the variable written by *d*
- It can be proven that the "smallest" sets IN[*n*] and OUT[*n*] that satisfy this system are exactly the solution for the Reaching Definitions problem
  – We will ignore: how do we know that this system has *any* solutions? how about a *unique smallest* one?

Iteratively Solving the System of Equations

OUT[*n*] = $\varnothing$ for each CFG node *n*

change = true

While (*change*)

1. For each *n* other than ENTRY and EXIT
   $OUT_{old}[n] = OUT[n]$
2. For each *n* other than ENTRY
   IN[*n*] = union of OUT[*m*] for all predecessors *m* of *n*
3. For each *n* other than ENTRY and EXIT
   $OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$
4. *change* = false
5. For each *n* other than ENTRY and EXIT
   If ($OUT_{old}[n]$!= OUT[*n*]) *change* = true

Worklist Algorithm

IN[n] = $\varnothing$ for all n

Put the successor of ENTRY on *worklist*

While (*worklist* is not empty)

1. Remove any CFG node m from the worklist
2. OUT[m] = (IN[m] – KILL[m]) $\cup$ GEN[m]
3. For each successor n of m

   old = IN[n]

   IN[n] = IN[n] $\cup$ OUT[m]

   If (*old* != IN[n]) add n to *worklist*

This is "chaotic" iteration

- The order of adding-to/removing-from the worklist is unspecified
  - e.g., could use stack, queue, set, etc.
- The order of processing of successor nodes is unspecified

Regardless of order, the resulting solution is always the same

A Simpler Formulation

- In practice, an algorithm will only compute IN[n]

$$\boxed{\mathrm{IN}[n] = \bigcup_{m \in \mathrm{Predecessors}(n)} (\mathrm{IN}[m] - \mathrm{KILL}[m]) \cup \mathrm{GEN}[m]}$$

  – Ignore predecessor *m* if it is ENTRY
- Worklist algorithm
  – IN[*n*] = ∅ for all *n*
  – Put the successor of ENTRY on the worklist
  – While the worklist is not empty, remove any *m* from the worklist; for each successors *n* of *m*, do
    - *old* = IN[*n*]
    - IN[*n*] = IN[*n*] ∪ (IN[*m*] – KILL[*m*]) ∪ GEN[*m*]
    - If (*old* != IN[*n*]) add *n* to *worklist*

A Few Notes

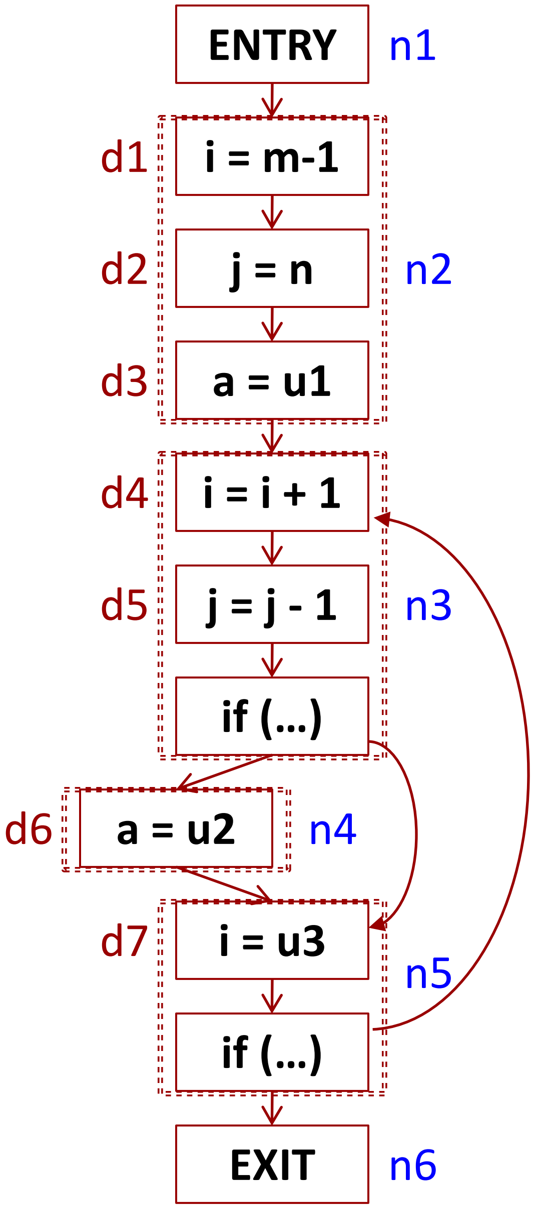
- We sometimes write

$$\boxed{\mathrm{IN}[n] = \bigcup_{m \in \mathrm{Predecessors}(n)} (\mathrm{IN}[m] \cap \mathrm{PRES}[m]) \cup \mathrm{GEN}[m]}$$

- PRES[*n*]: the set of all definitions “preserved” (i.e., not killed) by *n*; the complement of KILL[*n*]
- Efficient implementation: bitvectors
  - Sets are presented by bitvectors; set intersection is bitwise AND; set union is bitwise OR
  - GEN[*n*] and PRES[*n*] are computed once, at the very beginning of the analysis
  - IN[*n*] are computed iteratively, using a worklist

Reaching Definitions and Basic Blocks

- For space/time savings, we can solve the problem for basic blocks (i.e., CFG nodes are basic blocks)
  - Program points are before/after basic blocks
  - IN[n] is still the union of OUT[*m*] for predecessors *m*
  - OUT[*n*] is still (IN[*n*] – KILL[*n*]) ∪ GEN[*n*]
- $\mathrm{KILL}[n] = \mathrm{KILL}[s_1] \cup \mathrm{KILL}[s_2] \cup \ldots \cup \mathrm{KILL}[s_k]$
  - $s_1, s_2, \ldots, s_k$ are the statements in the basic blocks
- $\mathrm{GEN}[n] = \mathrm{GEN}[s_k] \cup (\mathrm{GEN}[s_{k-1}] - \mathrm{KILL}[s_k]) \cup (\mathrm{GEN}[s_{k-2}] - \mathrm{KILL}[s_{k-1}] - \mathrm{KILL}[s_k]) \cup \ldots \cup (\mathrm{GEN}[s_1] - \mathrm{KILL}[s_2] - \mathrm{KILL}[s_3] - \ldots - \mathrm{KILL}[s_k])$
  - GEN[*n*] contains any definition in the block that is downward-exposed (i.e., not killed by a subsequent definition in the block)



KILL[n2] = { d1, d2, d3, d4, d5, d6, d7 }
GEN[n2] = { d1, d2, d3 }
KILL[n3] = { d1, d2, d4, d5, d7 }
GEN[n3] = { d4, d5 }
KILL[n4] = { d3, d6 }
GEN[n4] = { d6 }
KILL[n5] = { d1, d4, d7 }
GEN[n5] = { d7 }

IN[n2] = { }
OUT[n2] = { d1, d2, d3 }

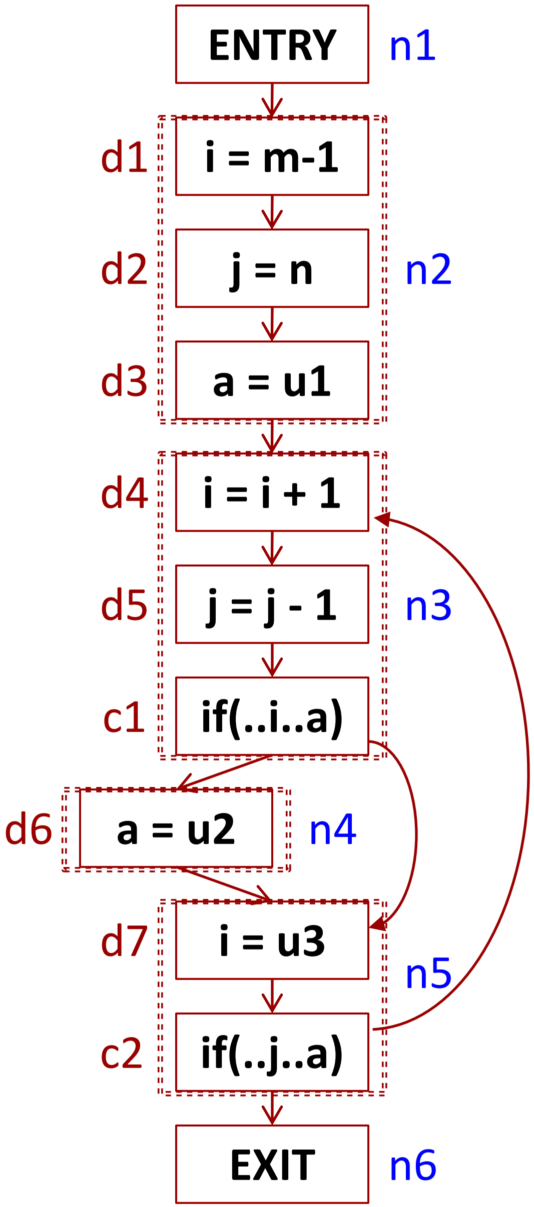
IN[n3] = { d1, d2, d3, d5, d6, d7 }
OUT[n3] = { d3, d4, d5, d6 }

IN[n4] = { d3, d4, d5, d6 }
OUT[n4] = { d4, d5, d6 }

IN[n5] = { d3, d4, d5, d6 }
OUT[n5] = { d3, d5, d6, d7 }

Uses of Reaching Definitions Analysis

- Def-use (du) chains
  - For a given definition (i.e., write) of a variable, which statements read the value created by the def?
  - For basic blocks: need all upward-exposed uses (use of variable does not have preceding def in the same basic block)
- Use-def (ud) chains
  - For a given use (i.e., read) of a variable, which statements performed the write of this value?
  - The reverse of du-chains
- Goal: potential write-read data dependences
  - Compiler optimizations
  - Program understanding (e.g., slicing)
  - Data-flow-based testing: coverage criteria
  - Semantic checks: e.g., use of uninitialized variables



<u>Upward exposed uses:</u>

USES[n2] = { m@d1, n@d2, u1@d3 }

USES[n3] = { i@d4, j@d5, a@c1 }

USES[n4] = { u2@d6 }

USES[n5] = { u3@d7, j@c2, a@c2 }

<u>Reaching definitions:</u>

IN[n3] = { d1, d2, d3, d5, d6, d7 }

IN[n4] = { d3, d4, d5, d6 }

IN[n5] = { d3, d4, d5, d6 }

<u>Def-use chains across basic blocks:</u>

DU[d1] = upward exposed uses of variable i in all basic blocks *n* such that d1 ∈ IN[*n*] = { i@d4 }

DU[d2] = { j@d5 }

DU[d3] = { a@c1, a@c2 }

DU[d4] = { }

DU[d5] = { j@d5, j@c2 }

DU[d6] = { a@c1, a@c2 }

DU[d7] = { i@d4 }

<u>Def-use chains inside basic blocks:</u>

DU[d4] = { i@c1 }

<u>Use-def chains:</u>

UD[m@d1]= { }

UD[n@d2]= { }

UD[u1@d3]= { }

UD[i@d4]= { d1,d7 }

UD[j@d5]= { d2,d5 }

UD[i@c1]= { d4 }

UD[a@c1]= { d3,d6 }

UD[u2@d6]= { }

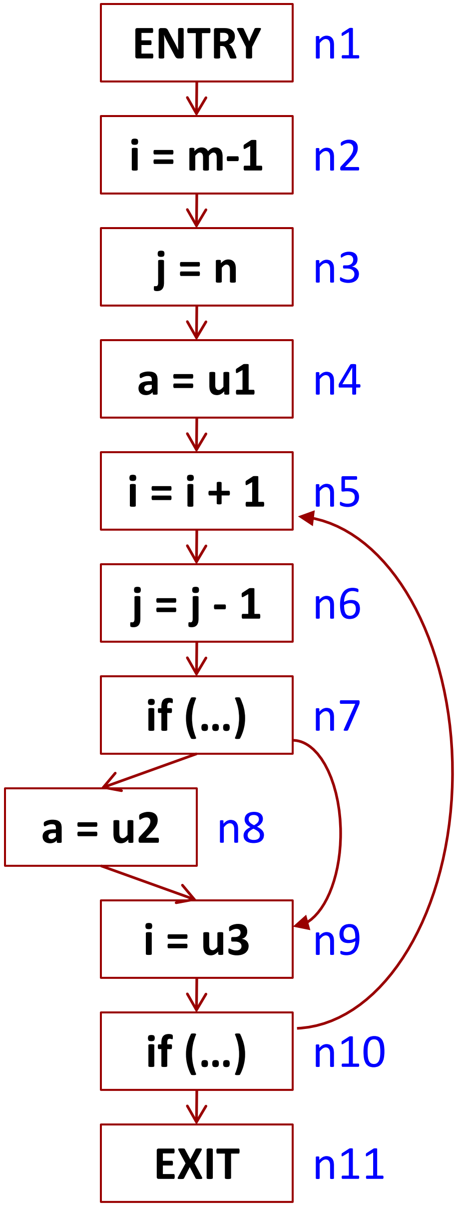
UD[u3@d7]= { }

UD[j@c2]= { d5 }

UD[a@c2]= { d3,d6 }

Example: Live Variables

- A variable *v* is **live** at a program point *p* if there exists a CFG path that
  - starts at *p*
  - ends immediately before some statement that reads *v*
  - does **not** contain a definition of *v*
- Thus, the value that *v* has at *p* could be used later
  - "could" because the CFG path may be infeasible
  - If *v* is not live at *p*, we say that *v* is **dead** at *p*
- For a CFG node *n*
  - IN[*n*] is the set of variables that are live at the program point immediately before *n*
  - OUT[*n*] is the set of variables that are live at the program point immediately after *n*



OUT[n1] = { m, n, u1, u2, u3 }
IN[n2] = { m, n, u1, u2, u3 }
OUT[n2] = { n, u1, i, u2, u3 }
IN[n3] = { n, u1, i, u2, u3 }
OUT[n3] = { u1, i, j, u2, u3 }
IN[n4] = { u1, i, j, u2, u3 }
OUT[n4] = { i, j, u2, u3 }
IN[n5] = { i, j, u2, u3 }
OUT[n5] = { j, u2, u3 }
IN[n6] = { j, u2, u3 }
OUT[n6] = { u2, u3, j }
IN[n7] = { u2, u3, j }
OUT[n7] = { u2, u3, j }
IN[n8] = { u2, u3, j }
OUT[n8] = { u3, j, u2 }
IN[n9] = { u3, j, u2 }
OUT[n9] = { i, j, u2, u3 }
IN[n10] = { i, j, u2, u3 }
OUT[n10] = { i, j, u2, u3 }
IN[n11] = { }

Examples of relationships:

OUT[n1] = IN[n2]

OUT[n7] = IN[n8] ∪ IN[n9]

IN[n10] = OUT[n10]

IN[n2] = (OUT[n2] – {i}) ∪ {m}

Formulation as a System of Equations

- For each CFG node *n*

$$\boxed{\mathrm{OUT}[n] = \bigcup_{m \in \mathrm{Successors}(n)} \mathrm{IN}[m]} \quad \boxed{\mathrm{IN}[\mathrm{EXIT}] = \varnothing}$$

$$\boxed{\mathrm{IN}[n] = (\mathrm{OUT}[n] - \mathrm{KILL}[n]) \cup \mathrm{GEN}[n]}$$

  - GEN[*n*] is the set of all variables that are read by *n*
  - KILL[*n*] is a singleton set containing the variable that is written by *n* (even if this variable is live immediately **after** *n*, it is not live immediately **before** *n*)
- The smallest sets IN[*n*] and OUT[*n*] that satisfy this system are exactly the solution for the Live Variables problem

Iteratively Solving the System of Equations

IN[*n*] = $\varnothing$ for each CFG node *n*

change = true

While (*change*)

1. For each *n* other than ENTRY and EXIT
   $\text{IN}_{\text{old}}[n] = \text{IN}[n]$
2. For each *n* other than EXIT
   OUT[*n*] = union of IN[*m*] for all successors *m* of *n*
3. For each *n* other than ENTRY and EXIT
   $\text{IN}[n] = (\text{OUT}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$
4. *change* = false
5. For each *n* other than ENTRY and EXIT
   If ($\text{IN}_{\text{old}}[n] \mathrel{!=} \text{IN}[n]$) *change* = true

Worklist Algorithm

OUT[n] = $\varnothing$ for all n

Put the predecessors of EXIT on *worklist*

While (*worklist* is not empty)

1. Remove any CFG node m from the worklist
2. IN[m] = (OUT[m] – KILL[m]) $\cup$ GEN[m]
3. For each predecessor n of m

    old = OUT[n]

    OUT[n] = OUT[n] $\cup$ IN[m]

    If (*old* != OUT[n]) add n to *worklist*

As with the worklist algorithm for Reaching Definitions, this is chaotic iteration. But, regardless of order, the resulting solution is always the same.

A Simpler Formulation

- In practice, an algorithm will only compute OUT[*n*]

$$\boxed{\mathrm{OUT}[n] = \bigcup_{m \in \mathrm{Successors}(n)} (\mathrm{OUT}[m] - \mathrm{KILL}[m]) \cup \mathrm{GEN}[m]}$$

  – Ignore successor *m* if it is EXIT

- Worklist algorithm
  – OUT[*n*] = ∅ for all *n*
  – Put the predecessors of EXIT on the worklist
  – While the worklist is not empty, remove any *m* from the worklist; for each predecessor *n* of *m*, do
    - *old* = OUT[*n*]
    - OUT[*n*] = OUT[*n*] ∪ (OUT[*m*] – KILL[*m*]) ∪ GEN[*m*]
    - If (*old* != OUT[*n*]) add *n* to *worklist*

A Few Notes

- We sometimes write

$$\boxed{\mathrm{OUT}[n] = \bigcup_{m \in \mathrm{Successors}(n)} (\mathrm{OUT}[m] \cap \mathrm{PRES}[m]) \cup \mathrm{GEN}[m]}$$

  - PRES[*n*]: the set of all variables "preserved" (i.e., not written) by *n*; the complement of KILL[*n*]
  - Efficient implementation: bitvectors
- Comparison with Reaching Definitions
  - Reaching Definitions is a forward data-flow problem and Live Variables is a backward data-flow problem
  - Other than that, they are basically the same
- Uses of Live Variables
  - Dead code elimination: e.g., when **x** is not live at **x=y+z**
  - Register allocation (more later …)

Example: Constant Propagation

- Can we guarantee that the value of a variable *v* at a program point *p* is always a known constant?
- Compile-time constants are useful
  - Constant folding: e.g., if we know that v is always 3.14 immediately before **w = 2*v**; replace it **w = 6.28**
    - Often due to symbolic constants
  - Dead code elimination: e.g., if we know that v is always false at **if (v) …**
  - Program understanding, restructuring, verification, testing, etc.

Basic Ideas

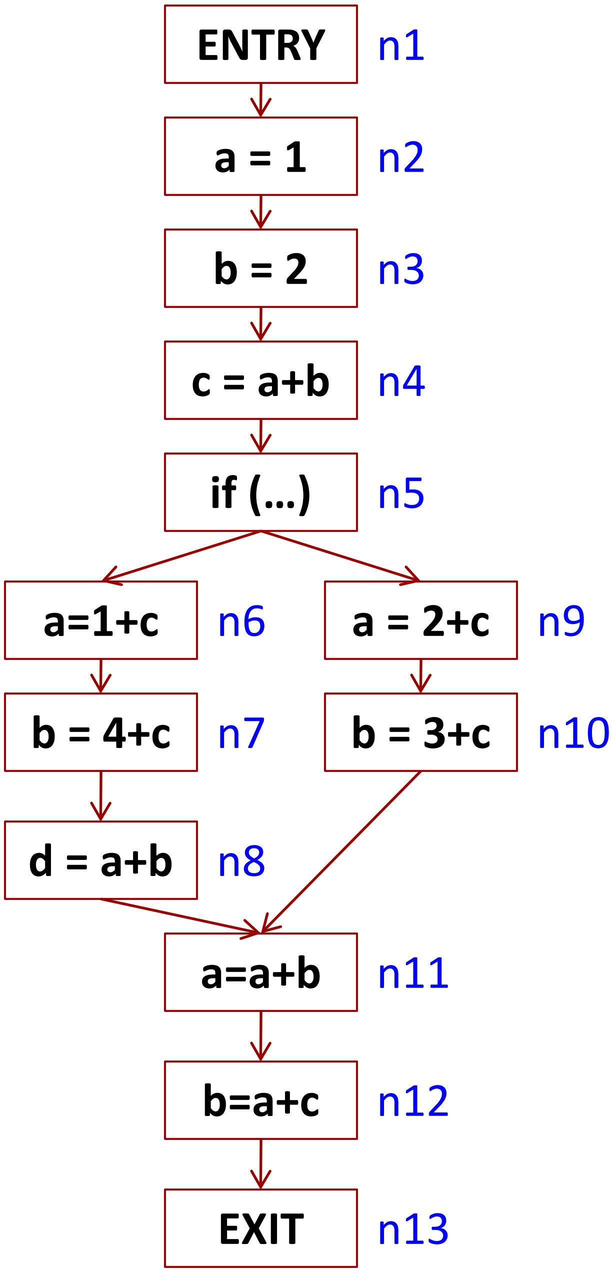
- At each CFG node *n*, IN[*n*] is a map Vars → Values
  - Each variable v is mapped to a value x ∈ Values
  - Values = all possible constant values ∪ { *nac* , *undef* }
- Special "value" *nac* (not-a-constant) means that the variable cannot be definitely proved to be a compile-time constant at this program point
  - E.g., the value comes from user input, file I/O, network
  - E.g., the value is 5 along one branch of an if statement, and 6 along another branch of the if statement
  - E.g., the value comes from some *nac* variable
- Special "value" *undef* (undefined): used temporarily during the analysis
  - Means "we have no information about v yet"

Formulation as a System of Equations

- OUT[ENTRY] = a map which maps each v to *undef*
- For any other CFG node *n*
  - IN[*n*] = Merge(OUT[*m*]) for all predecessors *m* of *n*
  - OUT[*n*] = Update(IN[*n*])
- Merging two maps: if v is mapped to c_1 and c_2 respectively, in the merged map v is mapped to:
  - If c_1 = *undef*, the result is c_2
  - Else if c_2 = *undef*, the result is c_1
  - Else if c_1 = *nac* or c_2 = *nac*, the result it *nac*
  - Else if $c_1 \neq c_2$, the result is *nac*
  - Else the result is c_1 (in this case we know that $c_1 = c_2$)

Formulation as a System of Equations

- Updating a map at an assignment **v = …**
  - If the statement is not an assignment, OUT[*n*] = IN[*n*]
- The map does not change for any w ≠ v
- If we have ***v = c***, where *c* is a constant: in OUT[*n*], v is now mapped to *c*
- If we have **v = p + q** (or similar binary operators) and IN[*n*] maps p and q to c_1 and c_2 respectively
  - If both c_1 and c_2 are constants: result is c_1+c_2
  - Else if either c_1 or c_2 is *nac*: result is *nac*
  - Else: result is *undef*



OUT[n1] = {a → *undef*, b → *undef*, c → *undef*, d → *undef* }
OUT[n2] = {a → *1*, b → *undef*, c → *undef*, d → *undef* }
OUT[n3] = {a → *1*, b → *2*, c → *undef*, d → *undef* }
OUT[n4] = {a → *1*, b → *2*, c → *3*, d → *undef* }

OUT[n6] = {a → *4*, b → *2*, c → *3*, d → *undef* }
OUT[n7] = {a → *4*, b → *7*, c → *3*, d → *undef* }
OUT[n8] = {a → *4*, b → *7*, c → *3*, d → *11* }

Merge

OUT[n9] = {a → *5*, b → *2*, c → *3*, d → *undef* }
OUT[n10] = {a → *5*, b → *6*, c → *3*, d → *undef* }

IN[n11] = {a → *nac*, b → *nac*, c → *3*, d → *11* }
OUT[n11] = {a → *nac*, b → *nac*, c → *3*, d → *11* }

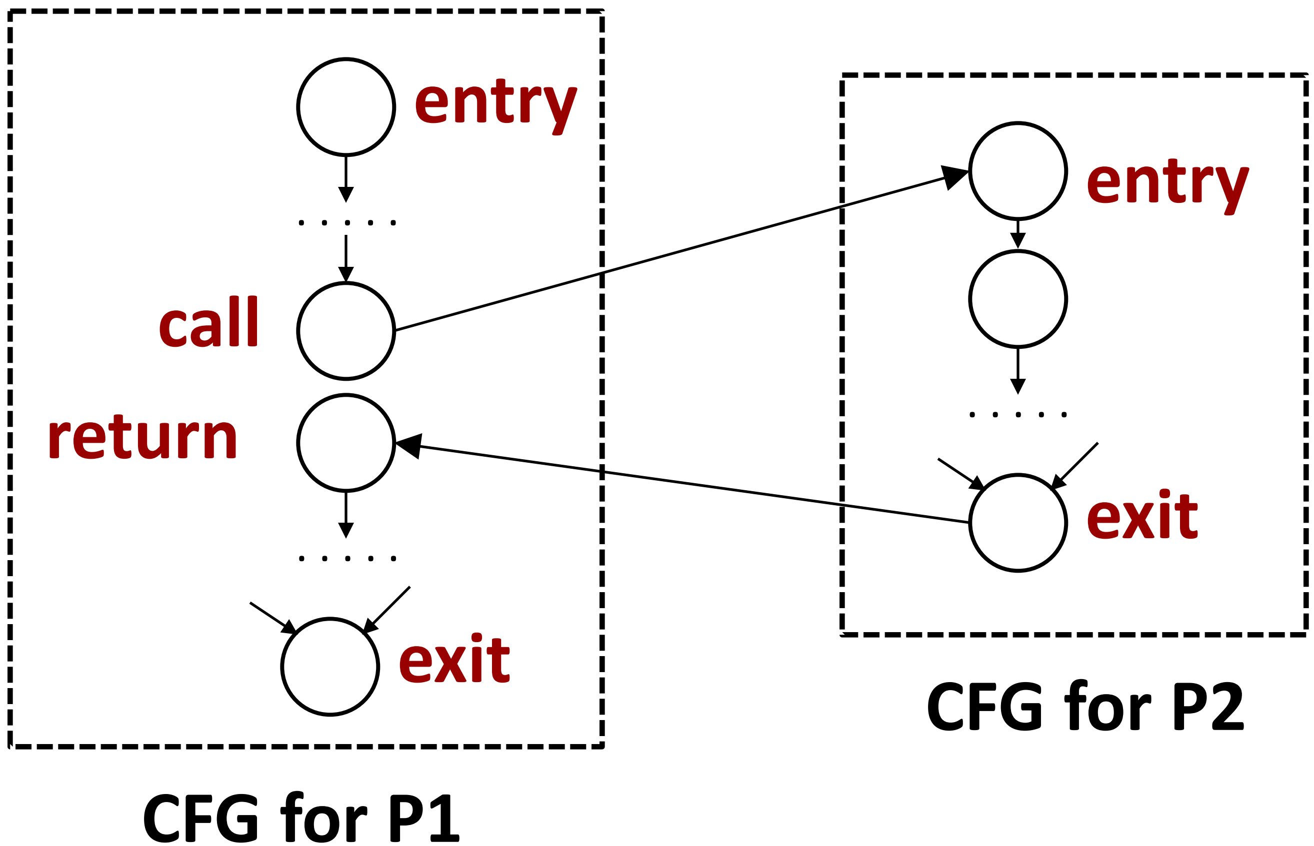
OUT[n12] = {a → *nac*, b → *nac*, c → *3*, d → *11* }

Note: in reality, d could be uninitialized at n11 and n12 (see Section 9.4.6 for a good discussion on this issue)

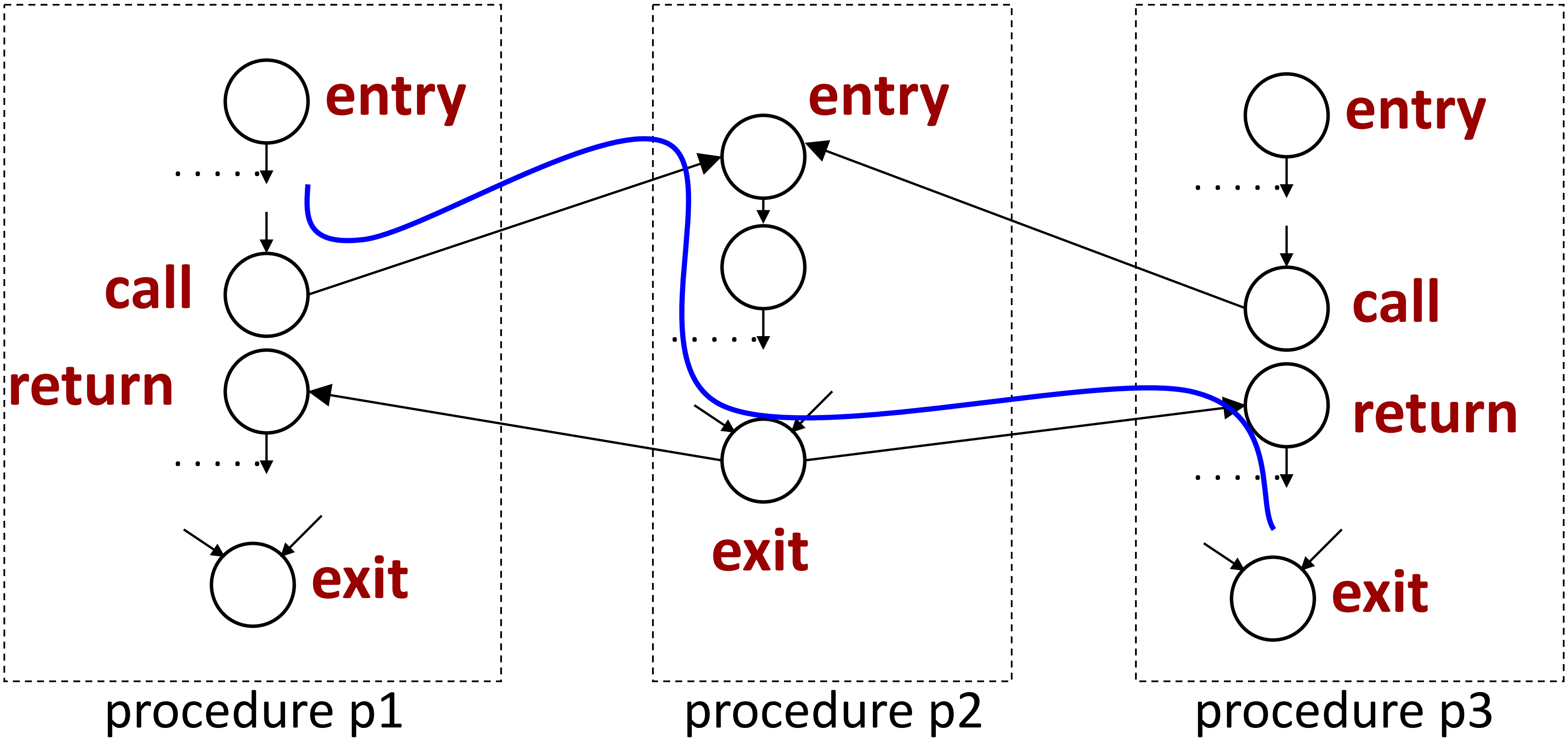
Example: Interprocedural Analysis

- CFG = procedure-level CFGs, plus (call,entry) and (exit,return) edges

```
void P1() {
    …
    P2();
    …
}
```



Valid Paths



Valid path: every (exit,return) matches the corresponding (call,entry)

The blue path is **not** valid

Design of **Inter**procedural Analysis

- **Intra**procedural analysis: separately considers the CFG for each procedure; makes conservative assumptions about any calls in the CFG
- **Inter**procedural analysis: considers all CFGs together; should consider all valid CFG paths
  - Option 1: do not distinguish between valid/invalid
    - Calling-context-insensitive analysis: does not keep track of the calling context of a procedure
    - Calling context example: the CFG **call node** that made the call (called "call site")
  - Option 2: calling-context-sensitive analysis
    - Keeps tracks of calling context, and avoids some of the invalid paths