

Dataflow Analysis

Chapter 9, Section 9.2, 9.3, 9.4

Dataflow Analysis

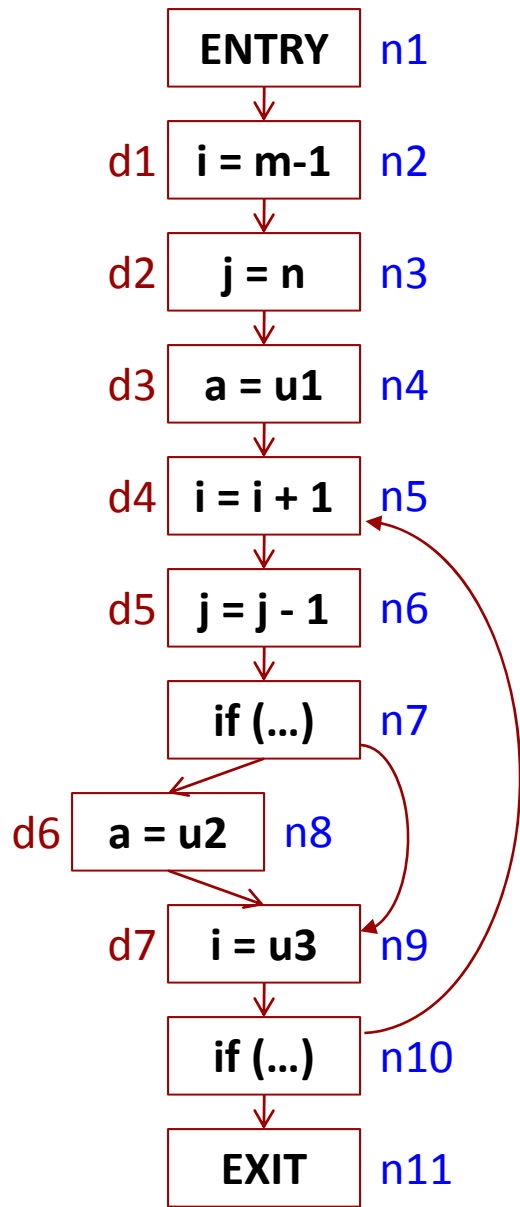
- Dataflow analysis is a sub-area of **static program analysis** (aka compile-time analysis)
 - Used in the compiler back end for optimizations of three-address code and for generation of target code
 - For software engineering: software understanding, restructuring, testing, verification
- Attaches 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**

Reaching Definitions

- A classical example of a dataflow analysis
 - We will consider **intraprocedural** analysis: only inside a single procedure, based on its CFG
- For a minute, 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 *infeasible* (could never occur)
 - Any compile-time analysis has to be *conservative*
- 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
 - Output of reaching definitions analysis: sets $IN[n]$ and $OUT[n]$ for each CFG node 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:
 $IN[n2] = OUT[n1]$
 $IN[n5] = OUT[n4] \cup OUT[n10]$
 $OUT[n7] = IN[n7]$
 $OUT[n9] = (IN[n9] - \{d1, d4\}) \cup \{d7\}$

Formulation as a System of Equations

- For each CFG node n

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

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

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

- $\text{GEN}[n]$ is a singleton set containing the definition d at n
- $\text{KILL}[n]$ is the set of all other definitions of the variable whose value is changed by d
- It can be proven that the “smallest” sets $\text{IN}[n]$ and $\text{OUT}[n]$ that satisfy this system are exactly the solution for the Reaching Definitions problem
 - To ponder: how do we know that this system has *any* solutions? how about a *unique smallest* one?

Iteratively Solving the System of Equations

$OUT[n] = \emptyset$ 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
 $IN[n] = \text{union of } OUT[m]$ 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 $(OUT_{old}[n] \neq OUT[n])$ $change = true$

Worklist Algorithm

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

Put the successor of ENTRY on *worklist*

While (*worklist* is not empty)

1. Remove a 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 ($old \neq IN[n]$) 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

8 Regardless of order, the resulting solution is always the same

A Simpler Formulation

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

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

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

A Few Notes

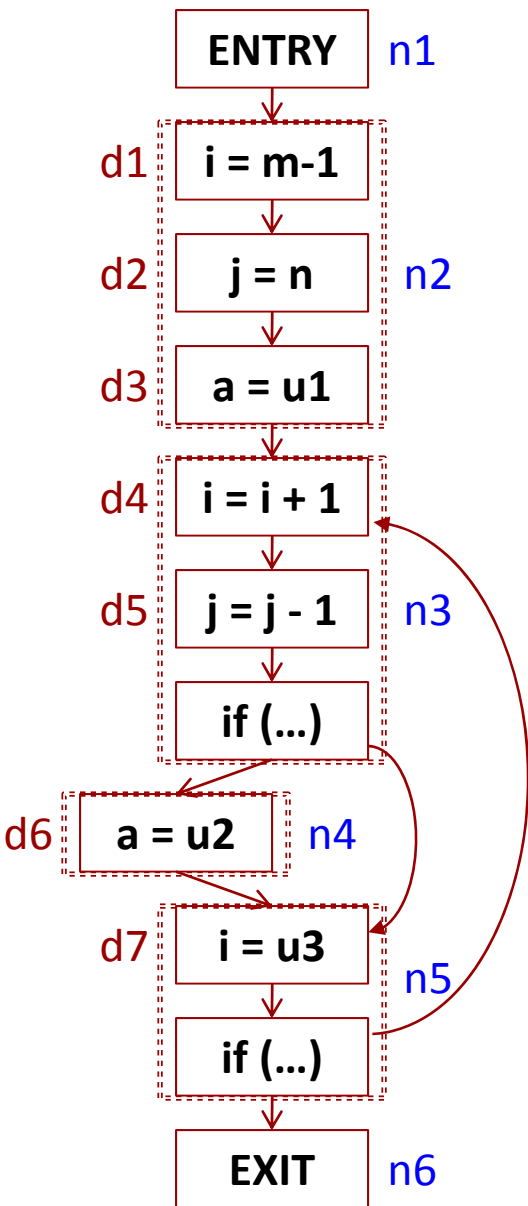
- We sometimes write

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

- $\text{PRES}[n]$: the set of all definitions “preserved” (i.e., not killed) by n
- Efficient implementation: bitvectors
 - Sets are presented by bitvectors; set intersection is bitwise AND; set union is bitwise OR
 - $\text{GEN}[n]$ and $\text{PRES}[n]$ are computed once, at the very beginning
 - $\text{IN}[n]$ 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 $(IN[n] - KILL[n]) \cup GEN[n]$
- $KILL[n] = KILL[s_1] \cup KILL[s_2] \cup \dots \cup KILL[s_k]$
 - s_1, s_2, \dots, s_k are the statements in the basic blocks
- $GEN[n] = GEN[s_k] \cup (GEN[s_{k-1}] - KILL[s_k]) \cup (GEN[s_{k-2}] - KILL[s_{k-1}] - KILL[s_k]) \cup \dots \cup (GEN[s_1] - KILL[s_2] - KILL[s_3] - \dots - KILL[s_k])$
 - $GEN[n]$ contains any definition in the block that is **downwards exposed** (i.e., not killed by a subsequent definition in the block)



KILL[n2] = { d4, d5, d6, d7 }

GEN[n2] = { d1, d2, d3 }

KILL[n3] = { d1, d2, d7 }

GEN[n3] = { d4, d5 }

KILL[n4] = { d3 }

GEN[n4] = { d6 }

KILL[n5] = { d1, d4 }

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 }

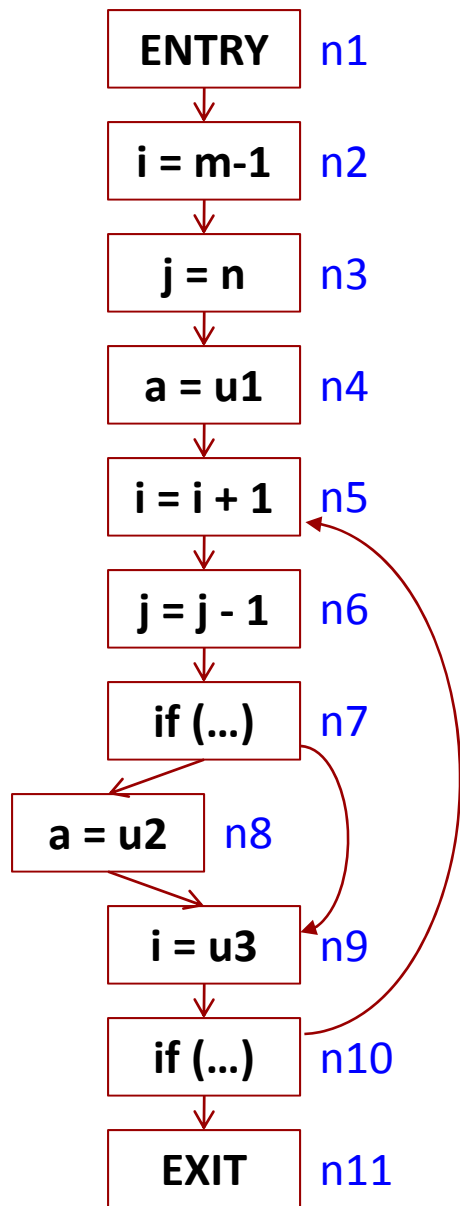
OUT[n5] = { d3, d5, d6, d7 }

Uses of Reaching Definitions Analysis

- Def-use (du) chains
 - For a given definition (i.e., **write**) of a memory location, which statements **read** the value created by the def?
 - For basic blocks: all upward-exposed uses
- Use-def (ud) chains
 - For a given use (i.e., **read**) of a memory location, which statements performed the **write** of this value?
 - The reverse of du-chains
- Goal: potential **write-read (flow) data dependences**
 - Compiler optimizations
 - Program understanding (e.g., slicing)
 - Dataflow-based testing: coverage criteria
 - Semantic checks: e.g., use of uninitialized variables
 - Could also find **write-write (output) dependences**

Live Variables

- A variable v is **live** at a program point p if there exists a CFG path that
 - starts at p
 - ends at a 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:

$OUT[n1] = IN[n2]$
 $OUT[n7] = IN[n8] \cup IN[n9]$
 $IN[n10] = OUT[n10]$
 $IN[n2] = (OUT[n2] - \{i\}) \cup \{m\}$

Formulation as a System of Equations

- For each CFG node n

$$\text{OUT}[n] = \bigcup_{m \in \text{Successors}(n)} \text{IN}[m]$$

$$\text{IN}[\text{EXIT}] = \emptyset$$

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

- $\text{GEN}[n]$ is the set of all variables that are **read** by n
- $\text{KILL}[n]$ 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 $\text{IN}[n]$ and $\text{OUT}[n]$ that satisfy this system are exactly the solution for the Live Variables problem

Iteratively Solving the System of Equations

$IN[n] = \emptyset$ for each CFG node n

change = true

While (*change*)

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

Worklist Algorithm

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

Put the predecessors of EXIT on *worklist*

While (*worklist* is not empty)

1. Remove a 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 ($old \neq OUT[n]$) 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 $\text{OUT}[n]$

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

- Ignore successor m if it is EXIT
- Worklist algorithm
 - $\text{OUT}[n] = \emptyset$ for all n
 - Put the predecessors of EXIT on the worklist
 - While the worklist is not empty, remove m from the worklist; for each predecessor n of m , do
 - $old = \text{OUT}[n]$
 - $\text{OUT}[n] = \text{OUT}[n] \cup (\text{OUT}[m] - \text{KILL}[m]) \cup \text{GEN}[m]$
 - If $(old \neq \text{OUT}[n])$ add n to *worklist*

A Few Notes

- We sometimes write

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

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

Constant Propagation Analysis

- Can we guarantee that the value of a variable v at a program point p is always a known constant?
- Compile-time constants are quite 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

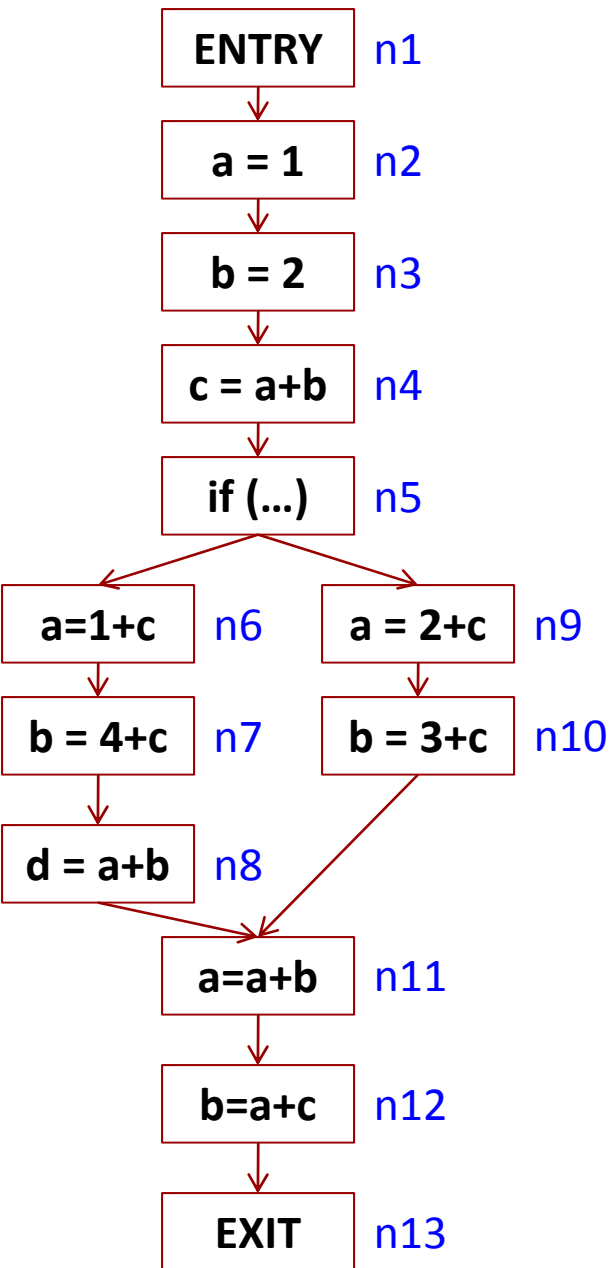
- At each CFG node n , $IN[n]$ is a map $Vars \rightarrow Values$
 - Each variable v is mapped to a value $x \in Values$
 - $Values =$ all possible constant values $\cup \{ nac, undef \}$
- Special “value” *nac* (not-a-constant) means that the variable cannot be definitely proved to be a compile-time constant at this program point
 - E.g., the value comes from user input, file IO, 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 – we have not seen values for v yet

Formulation as a System of Equations

- $\text{OUT}[\text{ENTRY}] =$ a map which maps each v to *undef*
- For any other CFG node n
 - $\text{IN}[n] = \text{Merge}(\text{OUT}[m])$ for all predecessors m of n
 - $\text{OUT}[n] = \text{Update}(\text{IN}[n])$
- **Merging** two maps: if v is mapped to c_1 and c_2 respectively, in the merged map v is mapped to:
 - If $c_1 = \text{undef}$, the result is c_2
 - Else if $c_2 = \text{undef}$, the result is c_1
 - Else if $c_1 = \text{nac}$ or $c_2 = \text{nac}$, the result is 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 $\mathbf{v} = \dots$
 - If the statement is not an assignment, $\text{OUT}[n] = \text{IN}[n]$
- The map does not change for any $w \neq v$
- If we have $\mathbf{v} = \mathbf{c}$, where \mathbf{c} is a constant: in $\text{OUT}[n]$, v is now mapped to \mathbf{c}
- If we have $\mathbf{v} = \mathbf{p} + \mathbf{q}$ (or similar binary operators) and $\text{IN}[n]$ maps p and q to \mathbf{c}_1 and \mathbf{c}_2 respectively
 - If both \mathbf{c}_1 and \mathbf{c}_2 are constants: result is $\mathbf{c}_1 + \mathbf{c}_2$
 - Else if either \mathbf{c}_1 or \mathbf{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 ↦ *nao*, b ↦ *nao*, c ↦ **3**, d ↦ **11**}

OUT[n11] = {a ↦ *nao*, b ↦ *nao*, c ↦ **3**, d ↦ **11**}

OUT[n12] = {a ↦ *nao*, b ↦ *nao*, 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)

Foundations of Dataflow Analysis

You need to understand the basic ideas, not
the technical details

Partial Order

- Given a set S , a **relation** r between elements of S is a set $r \subseteq S \times S$
 - Notation: if $(x,y) \in r$, write “ $x r y$ ”
 - Example: “less than” relation over integers
- A relation is a **partial order** if and only if
 - Reflexive: $x r x$
 - Anti-symmetric: $x r y$ and $y r x$ implies $x = y$
 - Transitive: $x r y$ and $y r z$ implies $x r z$
 - Example: “less than or equal to” over integers
 - By convention, the symbol used for a partial order is \leq or something similar to it (e.g. \sqsubseteq)

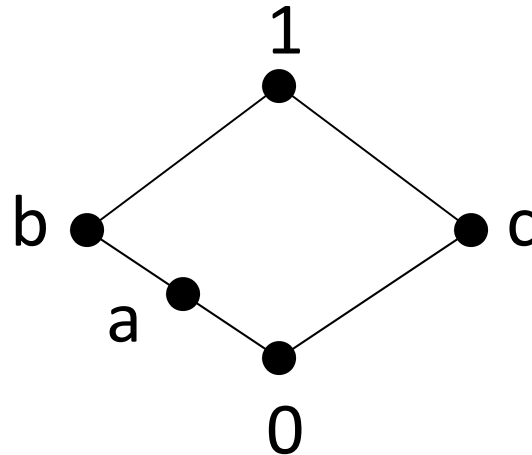
Partially Ordered Set

- **Partially ordered set** (S, \leq) is a set S with a defined partial order \leq
- Greatest element: x such that $y \leq x$ for all $y \in S$; often denoted by **1** or **T** (top)
- Least element: x such that $x \leq y$ for all $y \in S$; often denoted by **0** or **⊥** (bottom)
- It is not necessary to have 1 or 0 in a partially ordered set
 - e.g. $S = \{ a, b, c, d \}$ and only $a \leq b$ and $c \leq d$

Displaying Partially Ordered Sets

- Represented by an undirected graph
 - Nodes = elements of S
 - If $a \leq b$, a is shown below b in the picture
- If $a \leq b$, there is an edge (a,b)
 - But: transitive edges are typically not shown
- Example: $S = \{0,a,b,c,1\}$

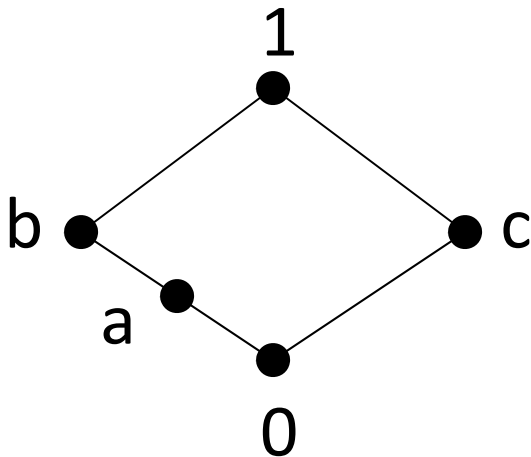
$$0 \leq a \leq b \leq 1$$
$$0 \leq c \leq 1$$



Implicit
transitive
edges:
 $0 \leq b$,
 $0 \leq 1, a \leq 1$

Meet

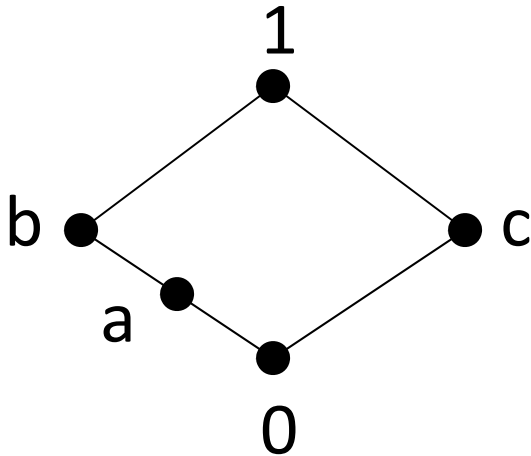
- S – partially ordered set, $a \in S$, $b \in S$
- A **meet** of a and b is $c \in S$ such that
 - $c \leq a$ and $c \leq b$
 - For any x : $x \leq a$ and $x \leq b$ implies $x \leq c$
 - Also referred to as “the greatest lower bound of a and b ”
 - Typically denoted by $a \wedge b$



$a \wedge b = a$	$a \wedge 0 = 0$
$a \wedge c = 0$	$a \wedge 1 = a$
$b \wedge c = 0$	$b \wedge 1 = b$
$b \wedge 0 = 0$	\dots

Join

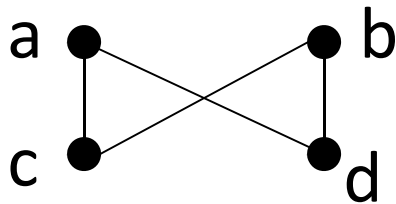
- A **join** of a and b is $c \in S$ such that
 - $a \leq c$ and $b \leq c$
 - For any x : $a \leq x$ and $b \leq x$ implies $c \leq x$
 - Also referred to as “the least upper bound of a and b ”
 - Typically denoted by $a \vee b$



$a \vee b = b$	$a \vee 0 = a$
$a \vee c = 1$	$a \vee 1 = 1$
$b \vee c = 1$	$b \vee 1 = 1$
$b \vee 0 = b$	\dots

Lattices

- Any pair (a,b) has either zero or one meets
 - Similarly for joins



$a \wedge b$ does not exist

“ $x \leq a$ and $x \leq b$ implies $x \leq \text{meet}$ ”: NO!

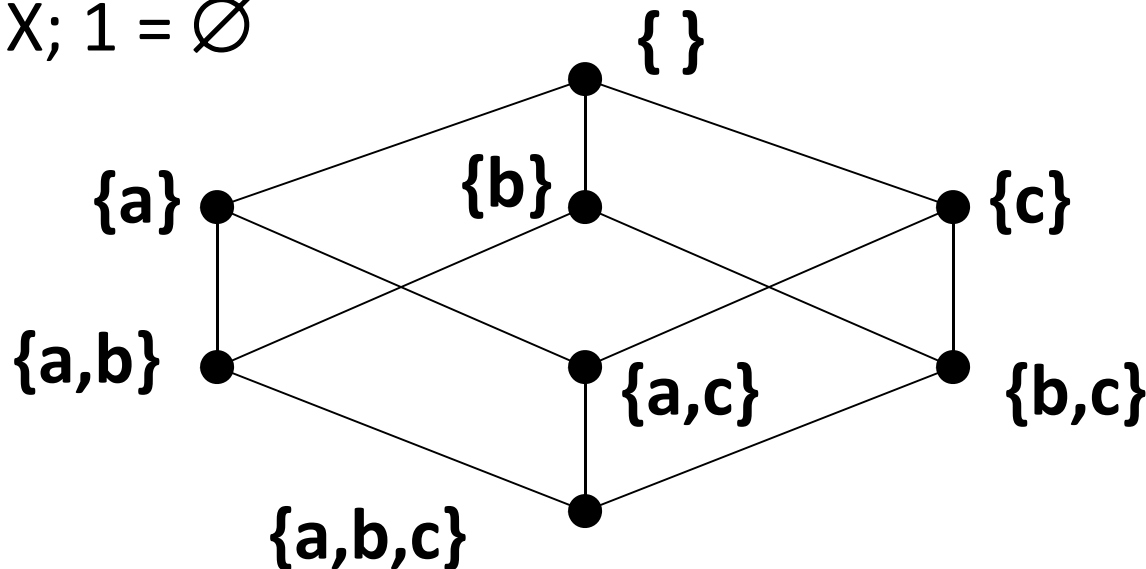
- If for every pair (a,b) there is a meet and a join, the partially ordered set is a **lattice**

So What?

- All of this is basic discrete math. What does it have to do with compile-time code analysis and code optimizations?
- For many analysis problems, **program properties** can be conveniently encoded as lattice elements
- If $a \leq b$, in some sense the property encoded by a is weaker (or stronger) than the one encoded by b
 - Exactly what “weaker”/“stronger” means depends on the problem

The Most Basic Lattice

- Many dataflow analyses use a lattice L that is the **power set $\mathcal{P}(X)$ of some set X**
 - $\mathcal{P}(X)$ is the set of all subsets of X
 - A lattice element is a subset of X
 - Partial order \leq is the \supseteq relation
 - Meet is set union \cup ; join is set intersection \cap
 - $0 = X$; $1 = \emptyset$



Reaching Definitions and Live Variables

- Let D be the set of all definitions in the CFG
- Reaching definitions: the lattice L is $\mathcal{P}(D)$
 - The solution for every CFG node is a lattice element
 - $IN[n] \in \mathcal{P}(D)$ is the set of definitions reaching n
 - The complete solution is a map $Nodes \rightarrow L$
- Let V be the set of all variables that are read anywhere in the CFG
- Live variables: the lattice L is $\mathcal{P}(V)$
 - The solution for every CFG node is a lattice element
 - $OUT[n] \in \mathcal{P}(V)$ is the set of variables live at n
 - The complete solution is a map $Nodes \rightarrow L$

The Role of Meet

- The partial order encodes some notion of strength for properties
 - if $x \leq y$, then x is “less precise” than y
- Reaching Definitions: $x \leq y$ iff $x \supseteq y$
 - x tells us that more things are possible, so x is less precise than y
 - Extreme case: if $x = 0 = D$, this tells us that any definition may reach
- $x \wedge y$ is less precise than x and y
 - greatest lower bound is the most precise lattice element that “describes” both x and y
 - E.g., the union of two sets of reaching definitions is the smallest (most precise) way to describe both

Transfer Functions

- A dataflow analysis defines a lattice L that encodes some program properties
- It also has to define **the effects of program statements** on these properties
 - A **transfer function** $f_n: L \rightarrow L$ is associated with each CFG node n
 - For forward problems: if the properties before the execution of n were encoded by $x \in L$, the properties after the execution of n are encoded by $f_n(x)$
- Reaching Definitions
 - $f_n(x) = (x \cap \text{PRES}[n]) \cup \text{GEN}[n]$
 - Expressed with meet and join: $f(x) = (x \vee a) \wedge b$

Intraprocedural Dataflow Analysis

- Given: an intraprocedural CFG, a lattice L , and transfer functions
 - Plus a lattice element $\rho \in L$ that describes the properties that hold at the entry node of the CFG
- The effects of one particular CFG path $p=(n_0, n_1, \dots, n_k)$ are

$$f_{n_k} (f_{n_{k-1}} (\dots f_1 (f_0 (\rho)) \dots))$$

- i.e., $\mathbf{f}_p(\rho)$, where \mathbf{f}_p is the composition of the transfer functions for nodes in the path
- n_0 is the entry node of the CFG

Intraprocedural Dataflow Analysis

- Analysis goal: for each CFG node n , compute a **meet-over-all-paths** solution

$$\text{MOP}(n) = \bigwedge_{p \in \text{Paths}(n_0, n)} f_p(\rho)$$

- **Paths**(n_0, n) the set of all paths from the entry node to n (the paths do not include n)
- This solution “summarizes” all properties that could hold immediately before n
 - Many execution paths: “meet” ensures that we get the greatest lower bound of their effects
 - E.g., the smallest set of reachable definitions

The MOP Solution

- The MOP solution encodes everything that could potentially happen at run time
 - e.g., for Reaching Definitions: if there exists a run-time execution in which variable x is assigned at m and read at n , set $\text{MOP}(n)$ is guaranteed to contain the definition of x at m
- Problems for computing $\text{MOP}(n)$:
 - Potentially infinite # paths due to loops
 - Even if there is a finite number of paths, there are too many of them: too expensive to compute $\text{MOP}(n)$ by considering each path separately

Approximating the MOP Solution

- A compromise: compute an **approximation** of the MOP solution
- A **correct** approximation: $S(n) \leq \text{MOP}(n)$
 - Recall that \leq means “less precise”
 - e.g., for Reaching Definitions $\text{IN}[n] \supseteq \text{MOP}(n)$
 - “safe solution” = “correct solution”
- A **precise** approximation: $S(n)$ should be as close to $\text{MOP}(n)$ as possible
 - In the best case, $S(n) = \text{MOP}(n)$

Standard Approximation Algorithm

- Idea: define a system of equations and then solve it with fixed-point computation

$$S(n) = \bigwedge_{m \in \text{Pred}(n)} f_m(S(m))$$

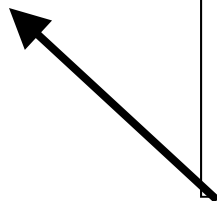
- This system has the form **$S = F(S)$**
 - **$S: \text{Nodes} \rightarrow L$** is map from CFG nodes to lattice elements
 - **$F: (\text{Nodes} \rightarrow L) \rightarrow (\text{Nodes} \rightarrow L)$** is a map from the old solution to the new one, based on the transfer functions **f_n**

Computing a Fixed Point

- Discrete math: if f is a function, a **fixed point** of f is a value x such that **$x = f(x)$**
 - We want to compute a fixed point of F
 - Standard algorithm (fixed-point computation)

at exit
 $S = \text{old_}S,$
so $S = F(S)$

```
S := [1,1,...,1]  
change := true  
while (change)  
    old_S := S;  
    S := F(S)  
    if (S ≠ old_S) change := true  
    else           change := false
```



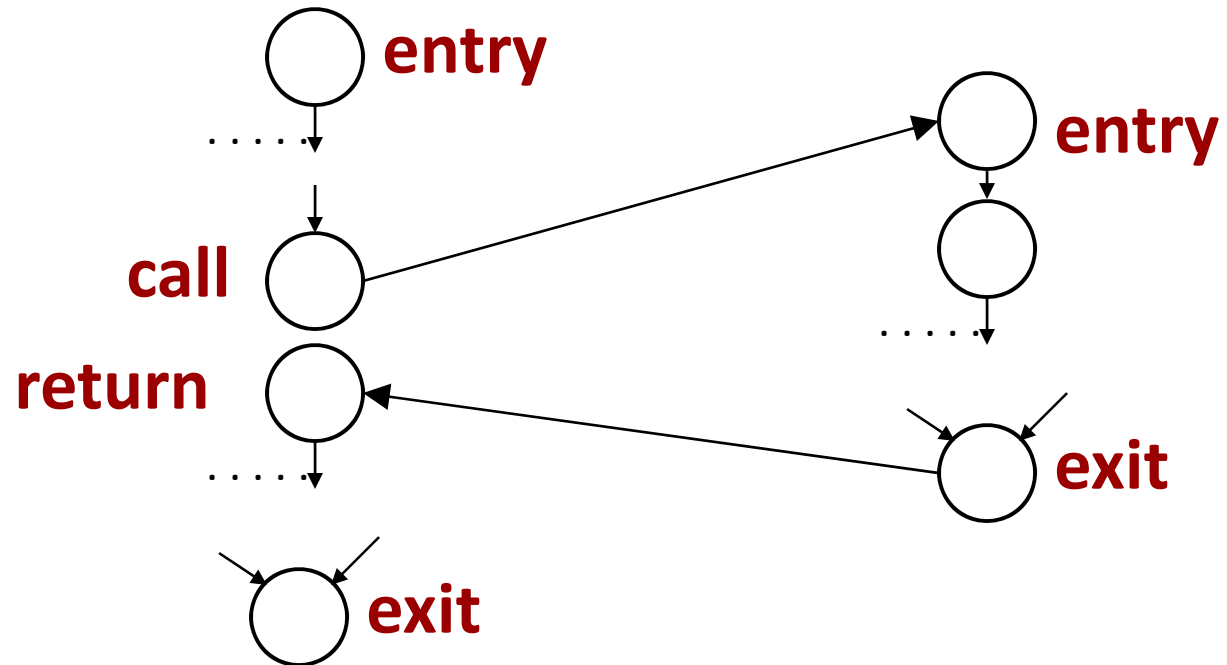
Does This Really Work?

- Does not necessarily terminate
- Common case: finite lattice + monotone transfer functions
 - monotone: $x \leq y$ implies $f(x) \leq f(y)$
- In this case, the algorithm provably terminates with a **safe approximation** of the MOP solution:
 $S(n) \leq MOP(n)$
 - For some categories of problems, the computed solution is **the same** as the MOP solution
 - e.g., for Reaching Definitions, but not for Constant Propagation

Interprocedural Dataflow Analysis

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

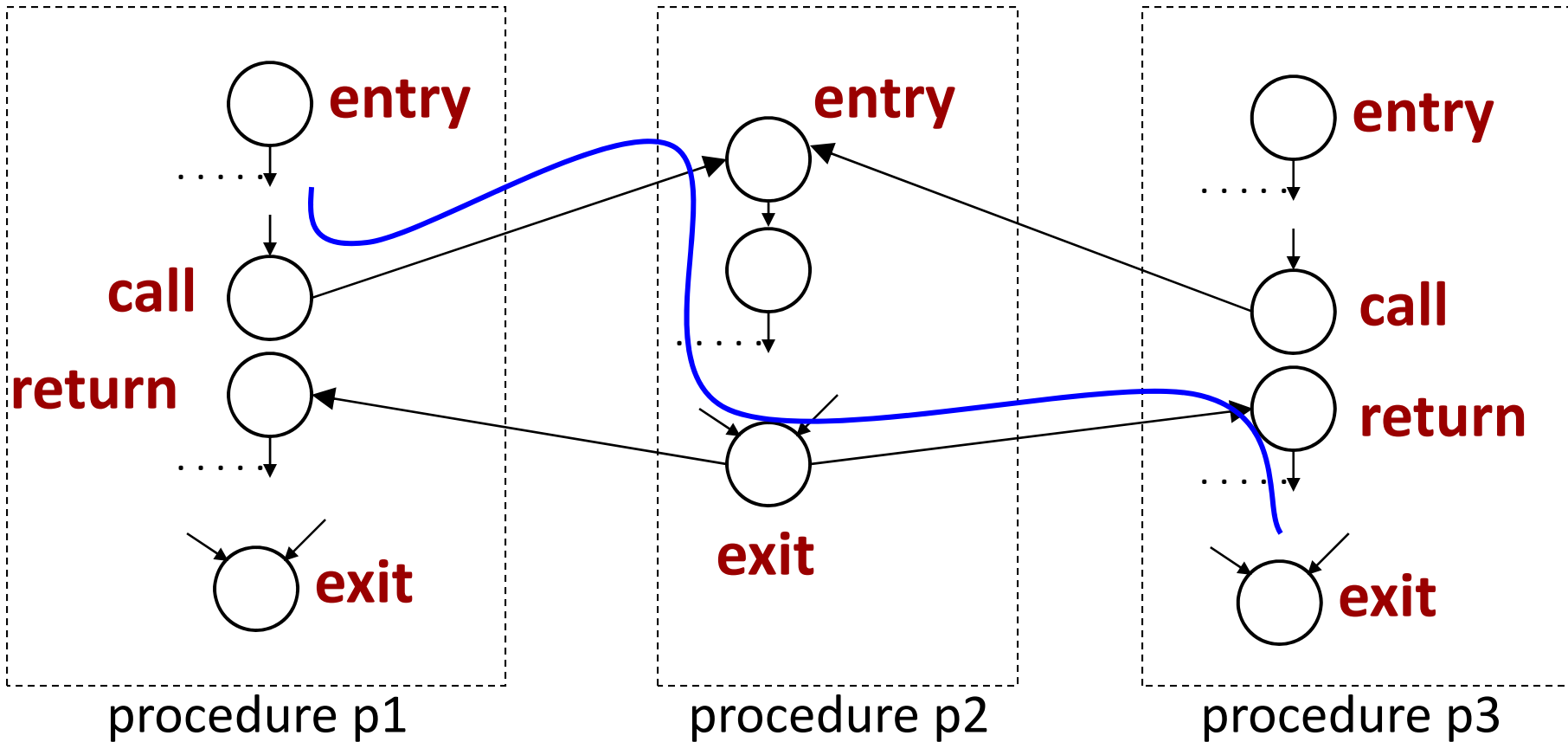
```
void n() {  
  ...  
  m();  
  ...  
}
```



Analysis Framework

- Again, define a lattice and transfer functions
 - e.g. the transfer functions at call nodes should describe the effects of parameter passing
- MOP is too imprecise here: not all paths in the CFG are feasible
 - **unrealizable paths** in the interprocedural CFG

Unrealizable Paths



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

Modified MOP Definition

- MORP: meet-over-all-realizable-paths
- Typically a **safe approximation** of MORP is computed (for efficiency)
 - Too many paths (even infinite # w/ recursion)
- Option 1: do not distinguish between realizable and unreliable paths
 - **Context-insensitive** analysis: does not keep track of the calling context of a procedure
- Option 2: **context-sensitive** analysis
 - Keeps tracks of calling context, and avoids some of the unrealizable paths