Dataflow Analysis

Dragon book, Chapter 9, Section 9.2, 9.3, 9.4

Dataflow Analysis

- Dataflow analysis is a sub-area of static program 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

Map of what is coming next

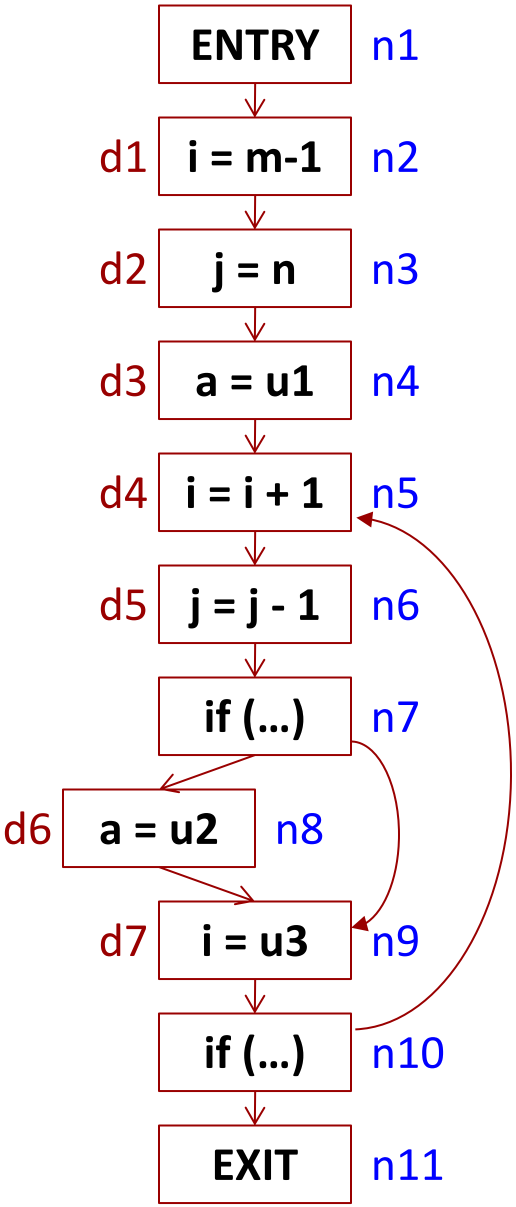
- Six intraprocedural dataflow analyses
  - Reaching Definitions
  - Live Variables
  - Copy Propagation
  - Available Expressions
  - Very Busy Expressions
  - Constant Propagation
  - Points-to Analysis
- Foundations of dataflow analysis
  - Framework: lattices and transfer functions
  - Meet-over-all-paths
  - Fixed point algorithms and solutions

Analysis 1: 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, assume 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*, 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: sets IN[*n*] and OUT[*n*] for each *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 other definitions of the variable whose value is changed 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
  - To ponder: how do we know that this system has *any* solutions at all? 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
   $OUT_{old}[n] = OUT[n]$
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
   $OUT[n] = (\ IN[n] - KILL[n]\ ) \cup GEN[n]$
4. *change* = false
5. For each n other than ENTRY
   If ($OUT_{old}[n]$!= $OUT[n]$) *change* = true

Questions

- What are the guarantees that this algorithm terminates?
- Does it compute a **correct** solution for the system of equations?
- Does it compute **the smallest** solution for the system of equations?
  - Assuming that there is a unique smallest solution
- How do we even know that this solution is the desired solution for Reaching Definitions?
- We will revisit these questions later, when considering the general machinery of dataflow analysis frameworks

Better Algorithm: Round-Robin, in Order

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

change = true

While (*change*)

change = false

For each n other than ENTRY, in rev. postorder

$\text{OUT}_{\text{old}}[n] = \text{OUT}[n]$

IN[n] = union of OUT[m] for all predecessors m of n

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

If ($\text{OUT}_{\text{old}}[n]$!= OUT[n]) *change* = true

Alternative: Worklist Algorithm

IN[n] = $\varnothing$ 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* != 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] = $\varnothing$ for all *n*
  – Put the successor of ENTRY on the worklist
  – While the worklist is not empty, remove *m* from the worklist; for each successor *n* of *m*, do
    - *old* = IN[*n*]
    - IN[*n*] = IN[*n*] $\cup$ (IN[*m*] – KILL[*m*]) $\cup$ GEN[*m*]
    - If (*old* != IN[*n*]) add *n* to *worklist*

A Few Notes

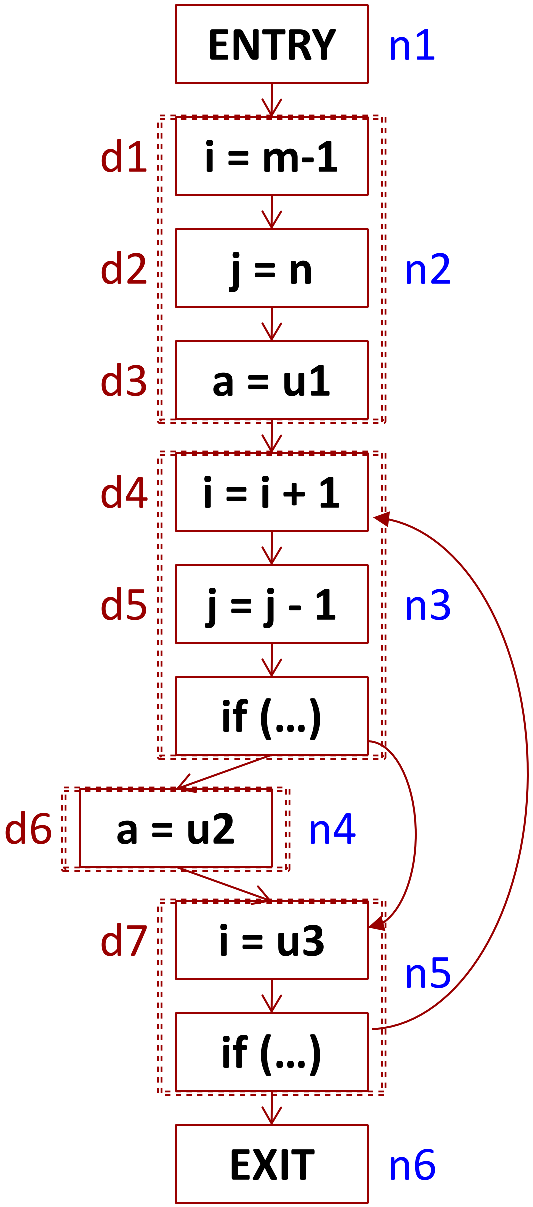
- We sometimes write

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

- PRES[*n*]: the set of all definitions "preserved" (i.e., not killed) by *n*
- 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 dataflow 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]) $\cup$ GEN[n]
- KILL[n] = KILL[s_1] $\cup$ KILL[s_2] $\cup$ … $\cup$ KILL[s_k]
  - s_1, s_2, …, 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]) $\cup$ … $\cup$ (GEN[s_1] – KILL[s_2] – KILL[s_3] – … – 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] = { 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 }

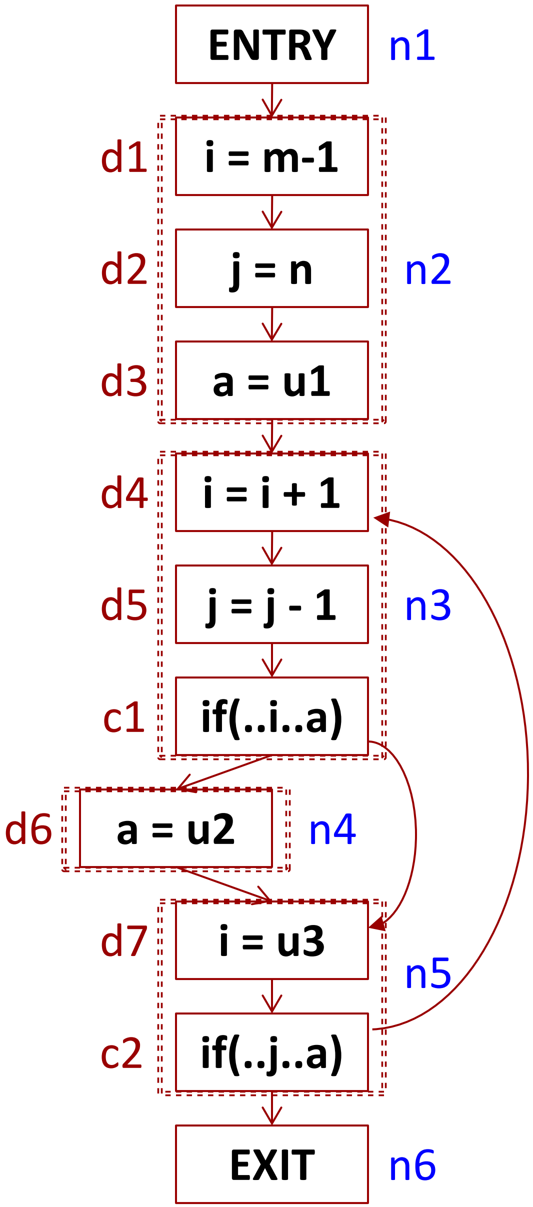
IN[n3] = { d1, d2, d3, d5, d6, d7 }
OUT[n3] = { d3, d4, d5, d6 }

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 memory location, which statements read the value created by the def?
  - For basic blocks: 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 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



Upward exposed uses:

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 }

Reaching definitions:

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

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

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

Def-use chains across basic blocks:

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 }

Def-use chains inside basic blocks:

DU[d4] = { i@c1 }

Use-def chains:

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]= { }

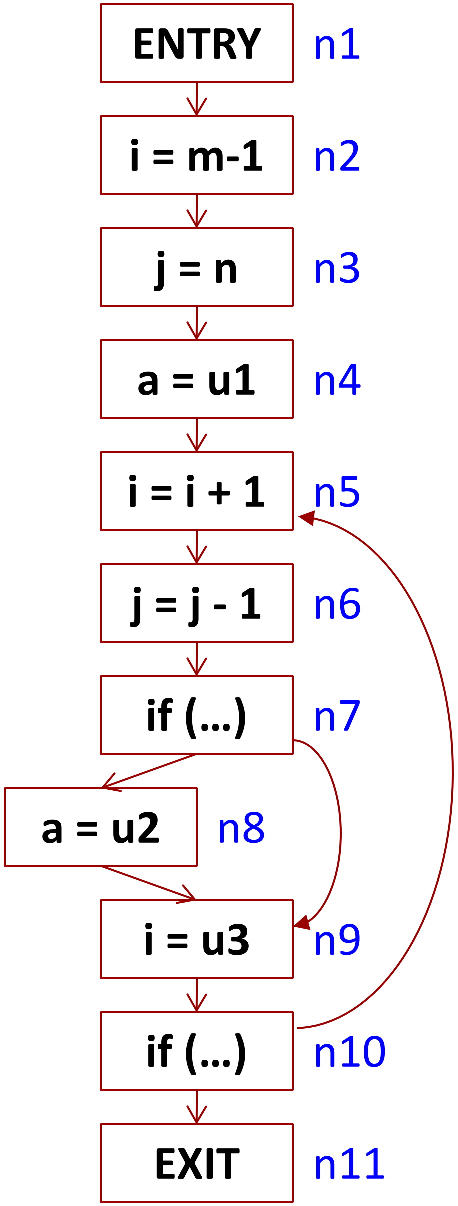
UD[u3@d7]= { }

UD[j@c2]= { d5 }

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

Analysis 2: 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] ∪ IN[n9]

IN[n10] = OUT[n10]

IN[n2] = (OUT[n2] – {i}) ∪ {m}

Formulation as a System of Equations

- For each CFG node *n*

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

$$\boxed{\mathrm{IN}[\mathrm{EXIT}] = \varnothing}$$

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

  - 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 EXIT
   $\mathrm{IN}_{old}[n] = \mathrm{IN}[n]$
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 EXIT
   $\mathrm{IN}[n] = (\ \mathrm{OUT}[n] - \mathrm{KILL}[n]\ ) \cup \mathrm{GEN}[n]$
4. *change* = false
5. For each *n* other than EXIT
   If ($\mathrm{IN}_{old}[n]$!= IN[*n*]) *change* = true

Better version: round-robin algorithm, in postorder

Worklist Algorithm

OUT[n] = $\varnothing$ 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* != 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 *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

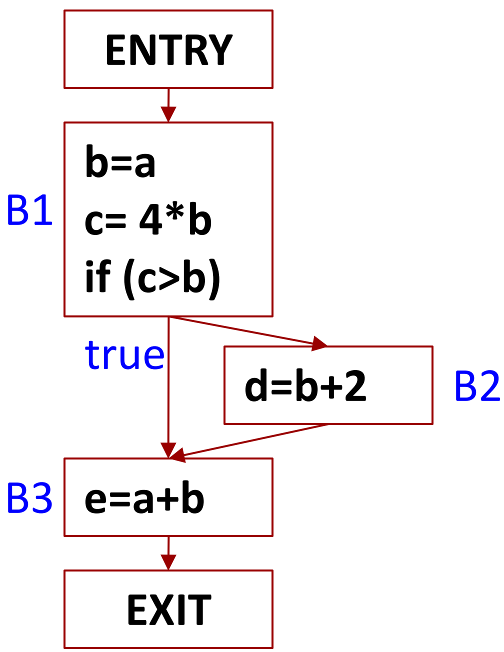
- We sometimes write

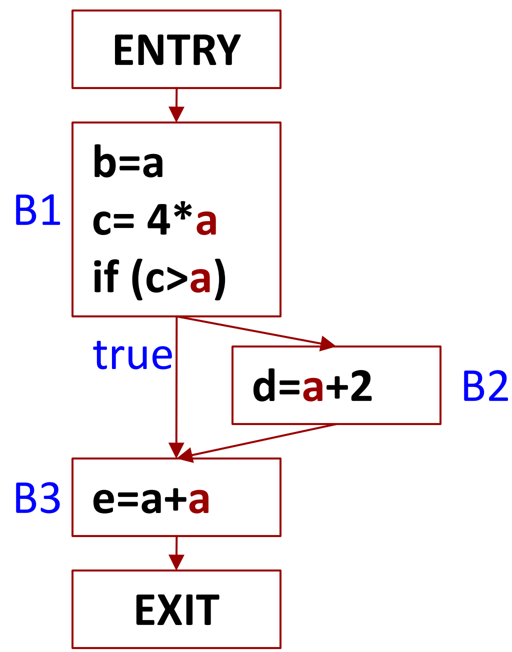
$$\boxed{\mathrm{OUT}[n] = \bigcup_{m \in \mathrm{Successors}(n)} (\mathrm{OUT}[m] \cap \mathrm{PRES}[m]) \cup \mathrm{GEN}[m]}$$

  – PRES[*n*]: the set of all variables “preserved” (i.e., not written) by *n*
  – 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 **x** is not live at **x=y+z**
  – Register allocation (more on this in CSE 756)

Analysis 3: Copy Propagation

- Copy propagation: for **x = y**, replace subsequent uses of **x** with **y**, as long as **x** and **y** have not changed along the way
  - Creates opportunities for dead code elimination: e.g., after copy propagation we may find that **x** is not live





1) Dead code elimination: **b=a**
2) Strength reduction: **e=a+a** use left shift instead of addition

Formulation as a System of Equations

- For each CFG node *n* (assume nodes = instructions)

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

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

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

  - IN[*n*] is a set of copy instructions **x=y** such that nether **x** nor **y** is assigned along any path from x=y to *n*
  - GEN[*n*] is
    - A singleton set containing the copy instruction, if *n* is a copy instruction
    - The empty set, otherwise
  - KILL[*n*]: if *n* assigns to **x**, kill every **y=x** and **x=y**
  - Note that we must use *intersection* of OUT[*m*]

Worklist Algorithm

IN[n] = *the set of all copy instructions*, 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] $\cap$ OUT[m]

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

In Reaching Definitions, we initialized IN[n] to the empty set; here we cannot do this, because of IN[n] = IN[n] $\cap$ OUT[m]

- Here the "meet" operator of the lattice is *set intersection*; the top element of the lattice is the set of all copy instructions
- In Reaching Definitions, "meet" is *set union*; "top" is the empty set

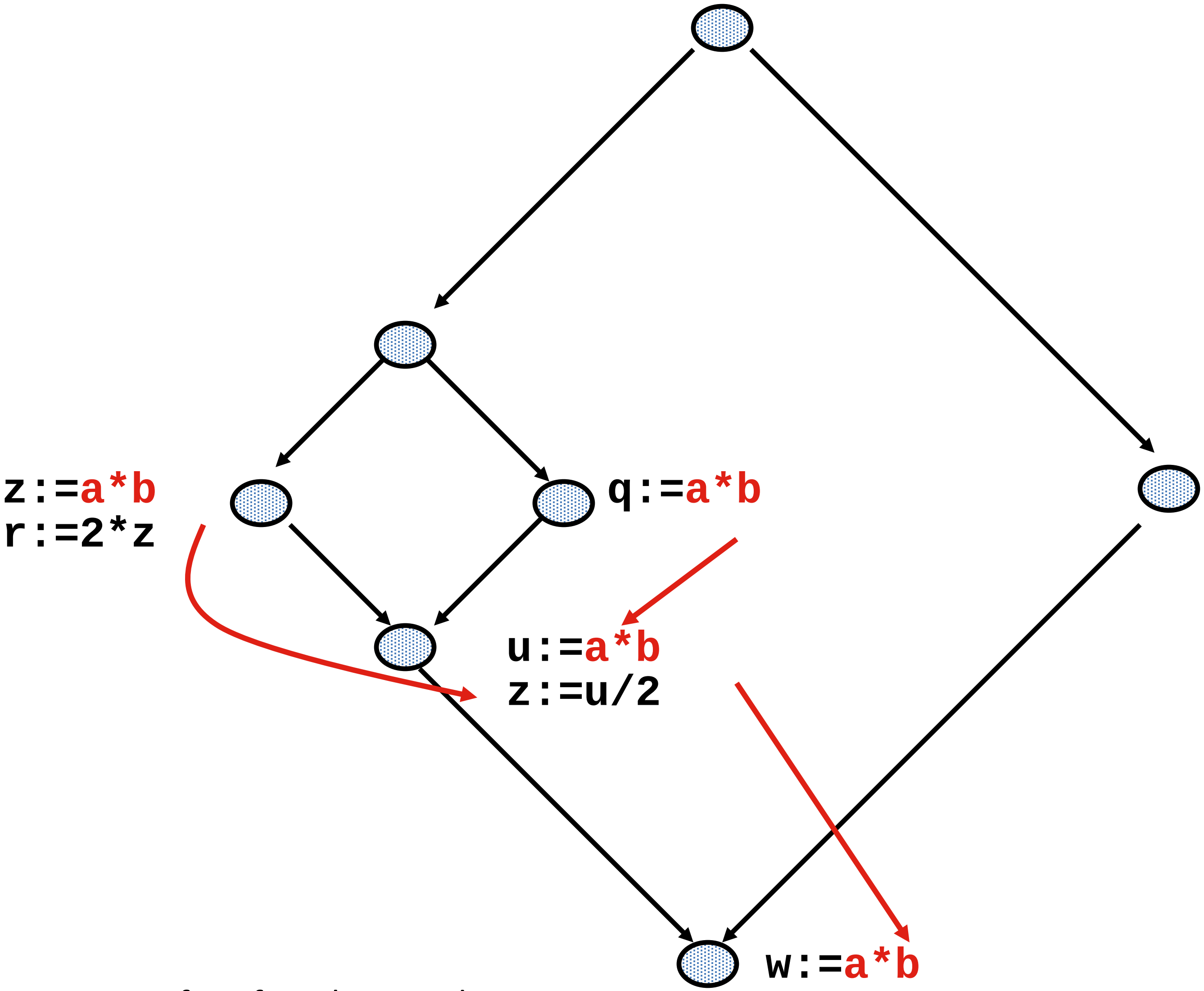
Classification

- Forward vs backward problems: intuitively, do we need to go forward along CFG paths, or backward?
  - Reaching Definitions: forward; Live Variables: backward; Copy Propagation: forward
- May vs must problems
  - Reaching Definitions: a definition may reach (union over predecessors – i.e., ∃ path …)
  - Live Variables: a use may be reached (union over successors – i.e., ∃ path …)
  - Copy Propagation: x and y must be preserved along all paths (intersection over predecessors – i.e., ∀ paths …)

Analysis 4: Available Expressions

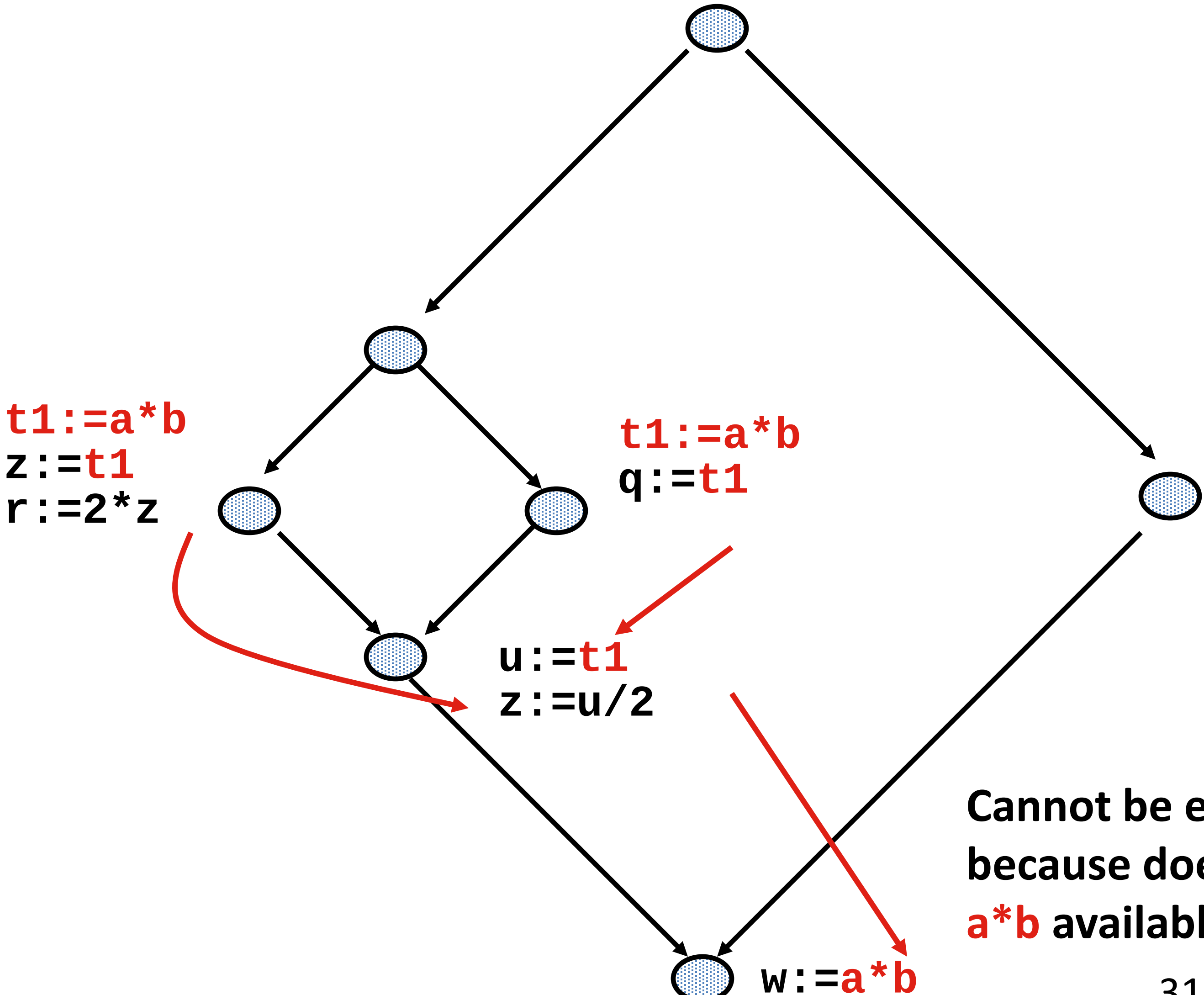
- Expression **x op y** is **available** at program point *p*
  1. Every path from ENTRY to *p* evaluates **x op y**
  2. After the last evaluation along the path, there are no subsequent assignments to **x** or **y**
- Useful for common subexpression elimination
- **Must** and **forward** problem
  - “Every path” – must problem
  - “From ENTRY to p” – forward problem

Common Subexpression Elimination



 Example courtesy of Prof. Barbara Ryder

Common Subexpression Elimination



Cannot be eliminated because does not have a*b available on all paths

 Example courtesy of Prof. Barbara Ryder

Formulation as a System of Equations

- For each CFG node *n*

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

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

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

  – IN[*n*] is a set of expressions **x op y** available at *n*
  – GEN[*n*] is
    - A singleton set containing the expression **x op y**, if *n* computes that expression
    - The empty set, otherwise
  – KILL[*n*]: if *n* assigns to **x**, kill every **x op y** and **y op x**
  – IN[*n*] is initialized to the set of all expressions appearing on the right-hand size of any instruction

Analysis 5: Very Busy Expressions

- Expression **x op y** is **very busy** at *p* if along every path from *p* we come to a computation of **x op y** before any redefinition of **x** or **y**
  - Useful for code motion: hoist **x op y** to program point *p*
  - **Backward must** problem

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

$$\boxed{\mathrm{OUT}[n] = \bigcap_{m \in \mathrm{Successors}(n)} \mathrm{IN}[m]}$$

  - Compare with Live Variables: backward **may** problem

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

Summary of Analyses 1-5

- Solution at a node is a **subset of a finite set** (thus, sometimes they are called "bitvector" problems)
- Functions are $\mathbf{f(x)=(A \cap x) \cup B}$ – "rapid" problems
  - Fast convergence w/ reverse postorder (forward analysis) or postorder (backward analysis): e.g.

    while (change)

    for each node n in reverse postorder

    IN[*n*] = … IN[m]…

    d+2 iterations; d is the max CFG loop nesting depth
  - If we use the worklist algorithm (i.e., chaotic iteration) non-determinism in worklist order and in order of successors

Analysis 6: 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 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

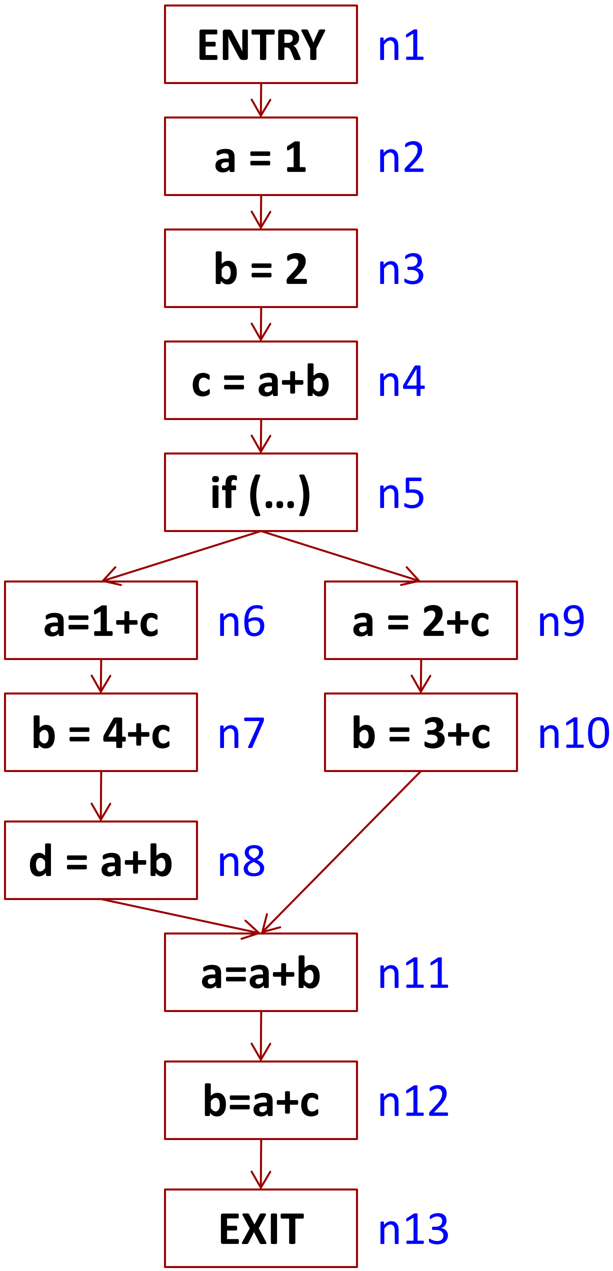
- At each CFG node *n*, IN[*n*] is a map Vars $\rightarrow$ Values
  - Each variable v is mapped to a value x $\in$ Values
  - Values = all possible constant values $\cup$ { *nac* , *undef* }
- Special "value" *nac* (not-a-constant) means that the variable cannot be definitely proved to be a compile-time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

Analysis 7: Points-To Analysis

- Question (oversimplified): can variable **x** contain the address of variable **y** at program point *p*?
- First abstraction: no arrays, no structs, no objects, no heap-allocated memory, no pointer arithmetic, no calls
- Instructions of interest
  - **x = &y**
  - **x = y**
  - **x = *y**
  - ***x = y**
  - **x = null**

Basic Ideas

- At each CFG node *n*, IN[*n*] is a set $\subseteq$ Vars × Vars
  - That is, a set of pairs of variables (**x**,**y**)
  - Alternative formulation: map Vars → PowerSet(Vars)
    - For each variable **x**, its points-to set Pt(**x**)
- If for some path from ENTRY to *n* the value of **x** is the address of **y** (when *n* is reached), then (**x**,**y**) must be an element of IN[*n*]
  - Often defined as "points-to graph": an edge **x** → **y** shows that **x** may point to **y**
- Similarly defined OUT[*n*]

Formulation as a System of Equations

- OUT[ENTRY] = empty set
- For any other CFG node *n*
  - IN[*n*] = Merge(OUT[*m*]) for all predecessors *m* of *n*
  - OUT[*n*] = Update(IN[*n*])
- Merging two points-to graphs: just the union of their edge sets

1. if (…) goto (4)	
2. x = &a	OUT[2] = { (x,a) }
3. goto (5)	
4. x = &b	OUT[4] = { (x,b) }
5. z = x	IN[5] = { (x,a), (x,b) }; OUT[5] = { (z,a), (z,b), (x,a), (x,b) }
6. w = &c	OUT[6] = { (z,a), (z,b), (x,a), (x,b), (w,c) }
7. *z = w	OUT[7] = { (z,a), (z,b), (x,a), (x,b), (w,c), (a,c), (b,c) }
8. v = *x	OUT[8] = { (z,a), (z,b), (x,a), (x,b), (w,c), (a,c), (b,c), (v,c) }

Formulation as a System of Equations

- Updating at an assignment **v = …** or ***v = …**
- **x = null**: OUT[*n*] = IN[*n*] – {x}×Vars
- **x = &y**: OUT[*n*] = (IN[*n*] – {x}×Vars) ∪ { **(x,y)** }
- **x = y**: OUT[*n*] = (IN[*n*] – {x}×Vars) ∪ { **(x,z)** | (y,z)∈ IN[*n*]}
- **x = *y**: OUT[n] = (IN[n] – {x}×Vars) ∪ { **(x,z)** | (y,w)∈ IN[*n*] ∧ (w,z) ∈ IN[*n*] }
- ***x = y**: OUT[n] = (IN[n] – nothing) ∪ { **(w,z)** | (x,w)∈ IN[*n*] ∧ (y,z) ∈ IN[*n*] }
  - Why not kill (w,…)? In general, we cannot assert that x *definitely* points to w, even if (x,w)∈ IN[*n*]; *more later …*

How About Real Programs?

- **x = malloc(…)** or **x = new X(…)**: artificial name $heap_i$: $OUT[n] = (IN[n] - \{x\} \times Vars) \cup \{\ \mathbf{(x,heap_i)}\ \}$
- **a[x] = y**: treat array **a** as one uniform block of data $OUT[n] = IN[n] \cup \{\ \mathbf{(a,z)} \mid (y,z) \in IN[n]\ \}$
- **x = a[y]**: $OUT[n] = (IN[n] - \{x\} \times Vars) \cup \{\ \mathbf{(x,y)} \mid (a,y) \in IN[n]\ \}$
- Fields of structs/objects: labels on points-to edges

struct S { int* f1; float* f2; };	
struct S* x = malloc(sizeof(struct S));	$(x,heap_1)$
(*x).f1 = &a; (*x).f2 = &b;	$(heap_1,f1,a)$ $(heap_1,f2,b)$
y = (*x).f1;	(y, a)

- Many complications: e.g., pointer arithmetic

Approximations

- **Flow-insensitive analysis**: ignore the flow of control and compute one points-to graph for the entire program (rather than a separate points-to graph for each CFG node)
- **Field-insensitive**: do not distinguish between fields

(*x).f1 = &a; (*x).f2 = &b; y = (*x).f1; treated as *x = &a; *x = &b; y = *x;

($heap_1$,f1,a) ($heap_1$,f2,b), (y,a) becomes ($heap_1$,a) ($heap_1$,b), (y,a), (y,b)

- **Base-object-insensitive**: treat (*x).f1 as f1

Java: x = new A; y = new A; x.f = new C; y.f = new D; z = y.f should lead to (x,$heap_1$), (y,$heap_2$), ($heap_1$,f,$heap_3$), ($heap_2$,f,$heap_4$), (z,$heap_4$)

Instead, it is treated as x = new A; y = new A; f = new C; f = new D; z = f and leads to (x,$heap_1$), (y,$heap_2$), (f,$heap_3$), (f,$heap_4$), (z,$heap_3$),(z,$heap_4$)

Flow-Insensitive Points-to Analysis

- A points-to graph could be $O(n^2)$ in size; a separate graph at each node is often too expensive
- "Fake" CFG with arbitrary sequences of statements

  while …

    switch ….

      case 1: statement 1

      case 2: statement 2 ….

- Points-to graph at the merge point of the switch
- Simplified functions without "kill" (more efficient):

$OUT[n] = (IN[n] - \{x\} \times Vars) \cup \ldots$ becomes

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

Loss of Precision: FI, FS, and Beyond

1. x = &a FS: OUT[1] = { (x,a) }
2. y = &b FS: OUT[2] = { (x,a), (y,b) }
3. z = &c FS: OUT[3] = { (x,a), (y,b), (z,c) }
4. *x = y FS: OUT[4] = { (x,a), (y,b), (z,c), (a,b) }
5. *a = … dependence between these statements:
6. … = c+1 FI: yes; FS: no
7. *x = z FS: OUT[7] = { (x,a), (y,b), (z,c), (a,b), (a,c) }
8. *a = … dependence between these statements:
9. … = b+2 FI and FS: yes (wrong!)

FI solution: (x,a), (y,b), (z,c), (a,b), (a,c)

Can we improve FS to eliminate (a,b) from OUT[7]?

FS with Strong Updates

- Updating at an assignment **v = …** or ***v = …**
  - If the statement is not an assignment, OUT[n] = IN[n]
- **x = …**: OUT[n] = (IN[n] – {x}×Vars) ∪ …
- ***x = y**: OUT[n] = (IN[n] – nothing) ∪ …
  - Why not kill (w,…) for when x points to w? In general, we cannot assert that x *definitely* points to w
- But what if the points-to set of x is a singleton set?
  - E.g., in the previous example, Pt(x) = { a }: can we kill (a,…) at *x = y?
  - If we can, OUT[7] will become { (x,a), (y,b), (z,c), (a,c) } and the precision is improved
    - False dependence between 8 and 9 disappears

FS with Strong Updates

- Proposal: at ***x = y**, if Pt(x) is a singleton set { w }, perform a strong update on w:
  – OUT[n] = (IN[n] – {w} ×Vars) ∪ …
- Not so fast … remember that w is just a static abstraction of a set of run-time memory locations; this set itself must be a singleton set

Example: recall field-insensitive analysis

x = malloc; (*x).f1 = &a; (*x).f2 = &b; y = (*x).f1; treated as x = &heap1, *x = &a; *x = &b; y = *x;

- FI without strong updates: at *x=&b, IN = { (x,$heap_1$), ($heap_1$,a)}, OUT = { (x,$heap_1$), ($heap_1$,a), ($heap_1$,b)} and later we get (y,a) , (y,b)
- With strong updates: OUT = { (x,$heap_1$),($heap_1$,b)} but (y,a) is lost!

“Dangerous” Strong Update

Which points-to graph node may correspond to multiple memory locations (and should **not** be strongly updated)?

- Array: one name for the entire array
- Local variable of a recursive procedures
- Dynamically allocated memory (even with field sensitivity)

curr = null

while (…) {

1. prev = curr IN[1] = {(prev,$heap_1$),(curr,$heap_1$),(y,$heap_2$),($heap_1$,fld,$heap_2$)}
2. curr = new X
3. y = new Y
4. curr.fld = y

} IN[5] = {(prev,$heap_1$),(curr,$heap_1$),(y,$heap_2$),($heap_1$,fld,$heap_2$)}

5. prev.fld = new Z OUT[5] = {(prev,$heap_1$),(curr,$heap_1$),(y,$heap_2$),($heap_1$,fld,$heap_3$)}
6. … curr.fld.fld2 … Dependence between these statements? **Yes**
7. … y.fld2 … With strong updates: **No**, because heap3.fld2 ≠ heap2.fld2

Foundations of Dataflow Analysis

Partial Order

- Given a set **S**, a relation **r** between elements of **S** is a set $\mathbf{r} \subseteq \mathbf{S} \times \mathbf{S}$
  - Notation: if $\mathbf{(x,y)} \in \mathbf{r}$, 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. □)

Partially Ordered Set

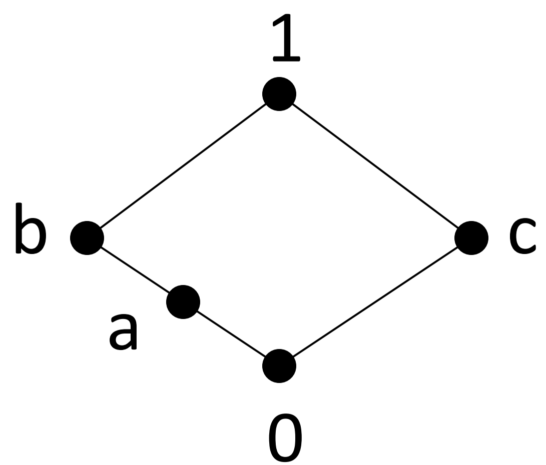
- Partially ordered set (S, ≤) is a set S with a defined partial order ≤
- Greatest element: x such that $y \leq x$ for all $y \in S$; often denoted by **1** or $\top$ (top)
- Least element: x such that $x \leq y$ for all $y \in S$; often denoted by **0** or $\bot$ (bottom)
- It is not necessary to have 1 or 0 in a partially ordered set
  - e.g. S = { a, b, c, d } and only $a \leq b$ and $c \leq d$
- We can always add an artificial top or bottom to the set (if we need one)

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: $S = \{0,a,b,c,1\}$

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

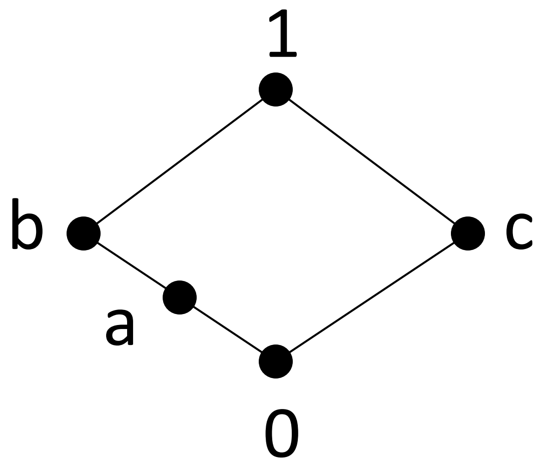
$0 \leq c \leq 1$



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

Meet

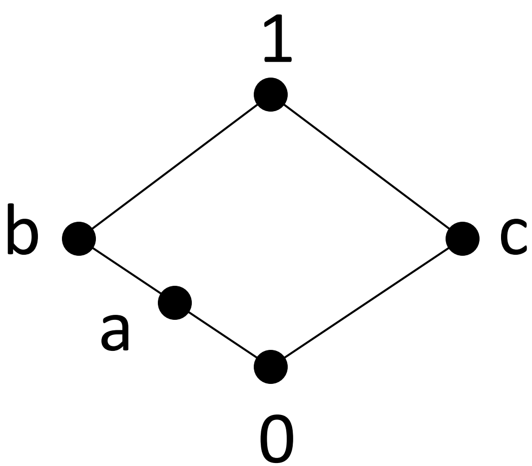
- S – partially ordered set, $a \in S$, $b \in S$
- A meet of a and b is $c \in S$ such that
  - $c \le a$ and $c \le b$
  - For any x: $x \le a$ and $x \le b$ implies $x \le c$
  - Also referred to as "the greatest lower bound of a and b"
  - Typically denoted by $a \wedge b$



$a \wedge b = a$	**$a \wedge 0 = 0$**
$a \wedge c = 0$	**$a \wedge 1 = a$**
$b \wedge c = 0$	**$b \wedge 1 = b$**
$b \wedge 0 = 0$	**. . . .**

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 V b



a V b = b   a V 0 = a

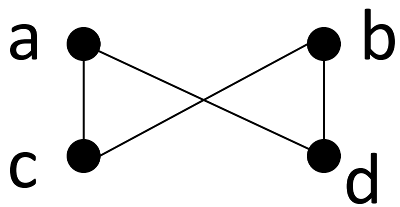
a V c = 1   a V 1 = 1

b V c = 1   b V 1 = 1

b V 0 = b   . . . .

Lattices

- Any pair (a,b) has either *zero* or *one* meets
  - Why can't there be two meets?
  - Similarly for joins



a **Λ** b does not exist
"x ≤ a and x ≤ b implies x ≤ meet": NO!

- If *every pair* (a,b) has is a meet and a join, the set is a **lattice** with operators **Λ** and **V**
  - If only a meet operator is defined: a **meet semilattice**
- Finite lattice: the underlying set is finite
- Finite-height lattice: any chain x < y < z < … is finite

Cross-Product Lattice

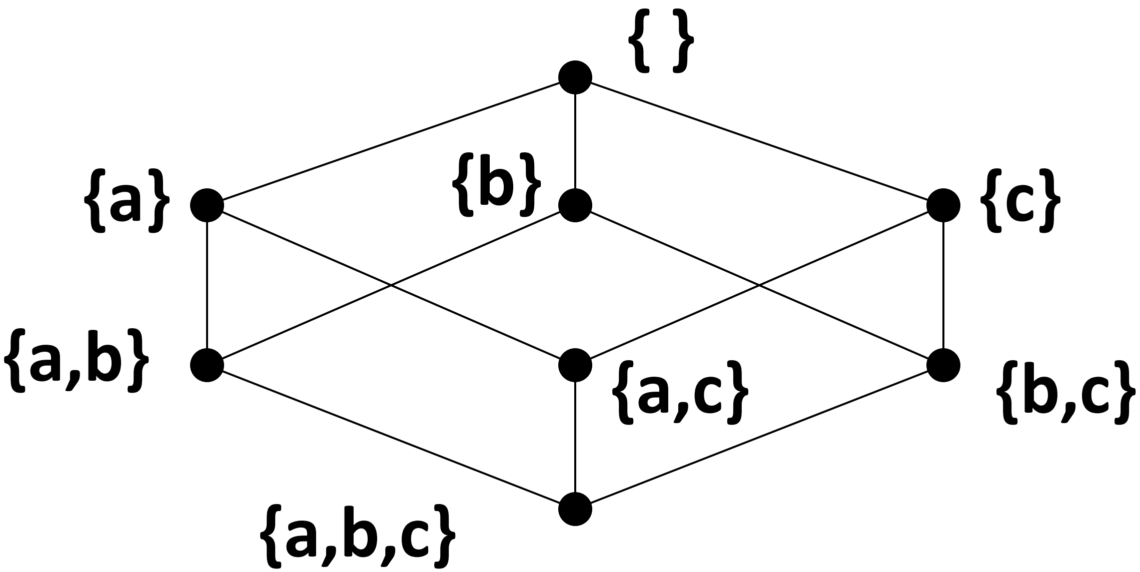
- Given a lattice $(L , \leq , \wedge , \vee)$
- Let $L^n = L \times L \times \ldots \times L$ (elements are n-tuples)
- Partial order: $(a_1,\ldots,a_n) \leq (b_1,\ldots,b_n)$ iff $a_i \leq b_i$ for all i
- Meet: $(a_1,\ldots,a_n) \wedge (b_1,\ldots,b_n) = (a_1 \wedge b_1,\ldots,a_n \wedge b_n)$
  - Same for join
- Cross-product lattice: $(L^n , \leq , \wedge , \vee)$
- If **L** has a bottom element **0**, L^n has a bottom element **(0,…,0)**
- If **L** has a top element **1**, L^n has a top element **(1,…,1)**
- If **L** has finite height, so does L^n

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 $a \le b$, 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
- We usually care only about "going in one direction" (down) in the lattice, so typically it is enough to have a **meet semilattice**

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 ≤ is the ⊇ relation
  - Meet is set union ∪; join is set intersection ∩
  - 0 = X; 1 = ∅



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 $Nodes \rightarrow L$
    - Actually, an element of the *cross-product lattice* $L^{|Nodes|}$; basically, an n-tuple
- 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
    - Any superset of the union has redundancy in it

The Role of Meet (cont'd)

- Recall the Constant Propagation problem
  - At each CFG node *n*, IN[*n*] is a map Vars $\rightarrow$ Values
  - Values = all possible constant values $\cup$ { *nac* , *undef* }
  - Values is an infinite lattice with finite height
    - *nac* $\leq$ any constant value $\leq$ *undef*
    - two different constant values are not comparable
- Meet operation in Values:
  - 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 $c_1 = c_2$)
- Problem lattice **L**: cross-product **Values**$^{|\mathrm{Vars}|}$

Transfer Functions

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

Function Space and Dataflow Framework

- Given: meet semilattice $(L,\leq,\wedge,1)$ with finite height
  - This is what we typically want as the part of the definition of the dataflow analysis
- A monotone functions space for L is a set F of functions $\mathbf{f : L \to L}$ such that
  - Each **f** is monotone: $\mathbf{x \leq y}$ implies $\mathbf{f(x) \leq f(y)}$
    - This is equivalent to $\mathbf{f(x \wedge y) \leq f(x) \wedge f(y)}$
  - F contains the identity function
  - F is closed under composition and meet: $\mathbf{f \circ g}$ and $\mathbf{f \wedge g}$ are in F [*Note:* $\mathbf{(f \circ g)(x) = f(g(x))}$ and $\mathbf{(f \wedge g)(x) = f(x) \wedge g(x)}$]
- Dataflow framework: **(L,F)**
  - Forward or backward; we will consider only forward
  - Framework instance (G,M): G=(N,E) is a CFG; M: N→F associates a transfer function $f \in F$ with each node $n \in N$

Intraprocedural Dataflow Analysis

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

$$\boxed{f_{n_k}(f_{n_{k-1}}(\ldots f_1(f_0(\eta))\ldots))}$$

  - i.e., **$f_p(\eta)$**, where $\mathbf{f}_{\boldsymbol{p}}$ 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

$$\mathbf{MOP}(n) = \bigwedge_{p \in \mathrm{Paths}(n_0,n)} f_p(\eta)$$

  – **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 MOP(*n*) is guaranteed to contain the definition of x at *m*
- Problems for computing 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 MOP(*n*) by considering each path separately
- Finding the MOP solution is undecidable for general monotone dataflow frameworks
  - Or even just for the constant propagation problem

Approximating the MOP Solution

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

Standard Approximation Algorithm

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

$$\mathbf{S(n) = \wedge_{m \in Pred(n)} \; f_{\mathit{m}}(S(m))}$$

- This system has the form **S = F(S)**
  - **S: Nodes → L** is map from CFG nodes to lattice elements (S is in the cross-product lattice $L^{|Nodes|}$)
  - **F: (Nodes → L) → (Nodes → L)** is a function that computes the new solution from the old one, based on the node-level transfer functions $\mathbf{f}_{n}$

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)

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

at exit

S=old_S,

so S=F(S)

Does This Really Work?

- Does not necessarily terminate
- Common case: finite-height lattice + monotone function space (as described earlier)
- In this case, the algorithm provably terminates with the greatest (maximum) fixed point MFP
  - Note: be careful with the difference between *maximal* (no one is > x) and *maximum* (x > everyone)
- MFP is a safe approximation of the MOP solution: MFP(n) ≤ 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

Outline of Proofs

- Termination with a fixed point
- monotonicity: $1^n \geq F(1^n) \geq F^2(1^n) \geq F^3(1^n) \geq \ldots$
- Finite height for L implies finite height for L^n, which gives us termination with $F^m(1^n) = F^{m+1}(1^n)$
  - $F^m(1^n)$ is a fixed point of F, and a solution to the system
- Is it the greatest (maximum) fixed point?
  - For any other fixed point S: $1^n \geq S$, $F(1^n) \geq F(S) = S, \ldots$
    - By induction on j, $F^j(1^n) \geq S$
- Why is MOP $\geq$ MFP?
  - For each CFG path $p=(n_0,n_1,\ldots,n_k)$, $\mathbf{f_p(\eta)} \geq$ MFP for any successor of n_k
  - Proof by induction on the length of paths

Distributive Frameworks

- Each **f** is monotone: **x ≤ y** implies **f(x) ≤ f(y)**
  - This is equivalent to **f(x Λ y) ≤ f(x) Λ f(y)**
- Distributive: **f(x Λ y) = f(x) Λ f(y)**
  - Each distributive function is also monotone
  - Examples: Reaching Defs, Live Variables, Available Expressions, Very Busy Expressions, Copy Propagation
- In this case, MFP = MOP
  - Proof outline: Since we already know that MOP ≥ MFP, enough to show that MFP ≥ MOP
  - Show by induction on j that $F^j(1^n) \geq$ MOP
  - Enough to show that F(MOP) = MOP: that is, MOP(n) = meet of f_m(MOP(m)) over all predecessors m of n
  - By definition, MOP(m) is a meet over all paths leading to m; f_m(meet of paths) = meet(f_m(path))

An Approximation: Flow-Insensitive Analysis

- Some problems are too complex/expensive to compute a solution specific to each CFG node
  - Typical example: pointer analysis (more later)
- Approximation: “pretend” that statements can execute in any order
  - Not only in the order defined by CFG paths
- Completely ignore all CFG edges – just consider the transfer functions at nodes
  - For technical reasons, make the functions “non-kill”: **$f(x) \leq x$** [e.g. as if KILL set was empty for Reaching Defs]
- Single solution (lattice element) for the entire CFG
- Naïve algo: start from **1** and apply the transfer functions in arbitrary order; get to a fixed point