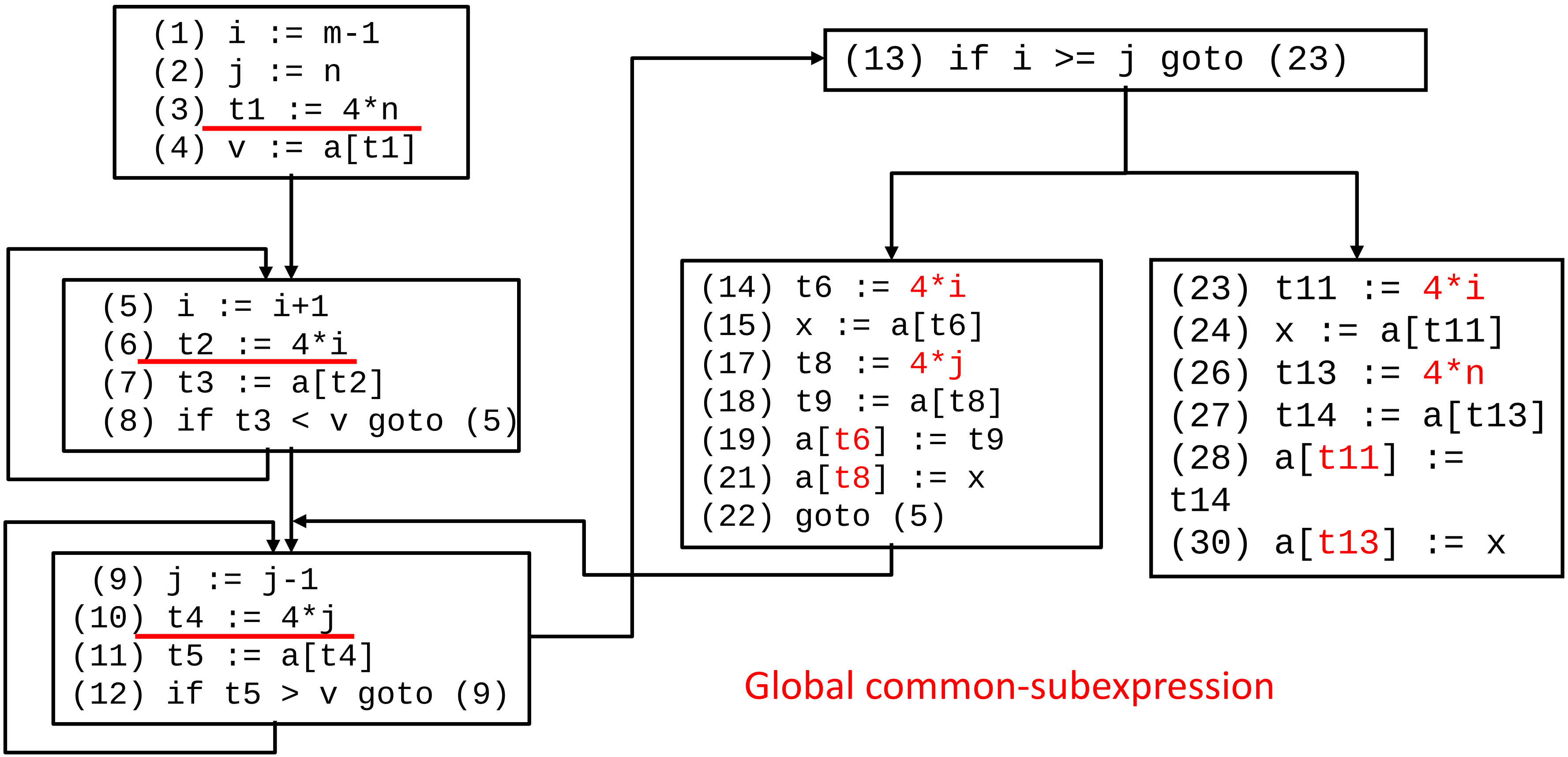
Dataflow Analysis

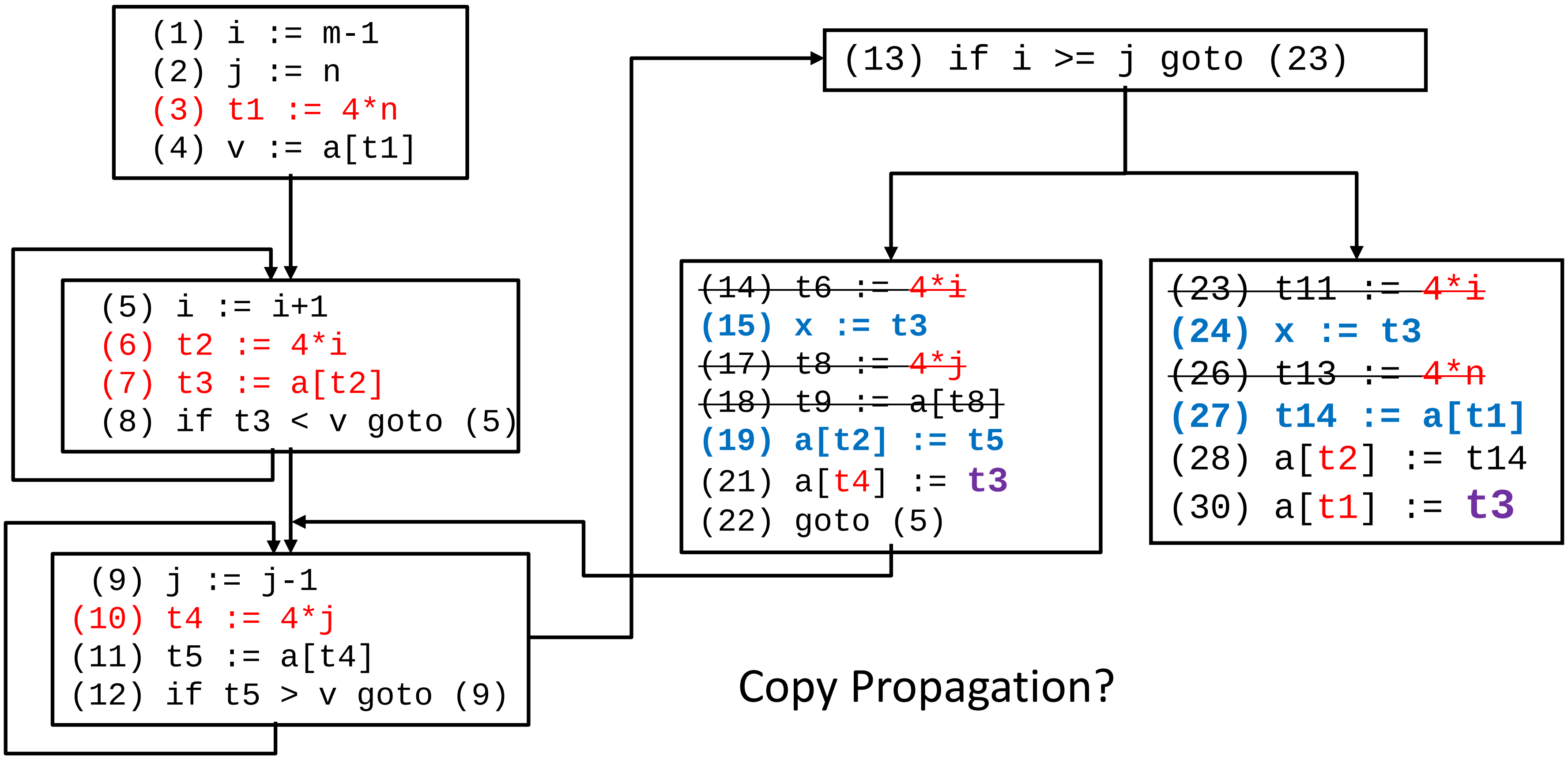
Local common-subexpression elimination and dead code elimination



Global common-subexpression

i@14 and 6 use the same definition?

Copy Propagation

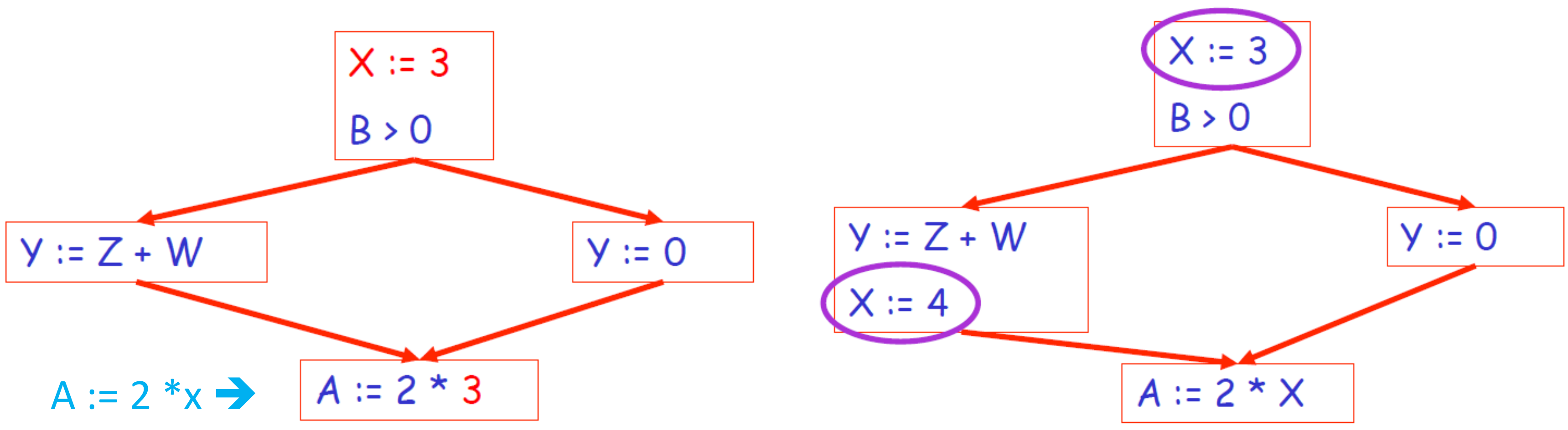


Copy Propagation?

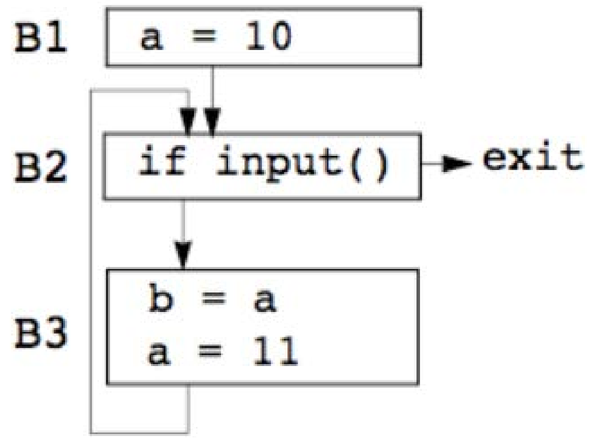
x@21 uses the definition @15?

Dataflow Analysis

- Data flow analysis:
  - Flow-sensitive: sensitive to the control flow in a function
  - Intra-procedural analysis, i.e., context-insensitive
  - Path-insensitive
- All the optimizations depend on dataflow analysis
- E.g. copy propagation

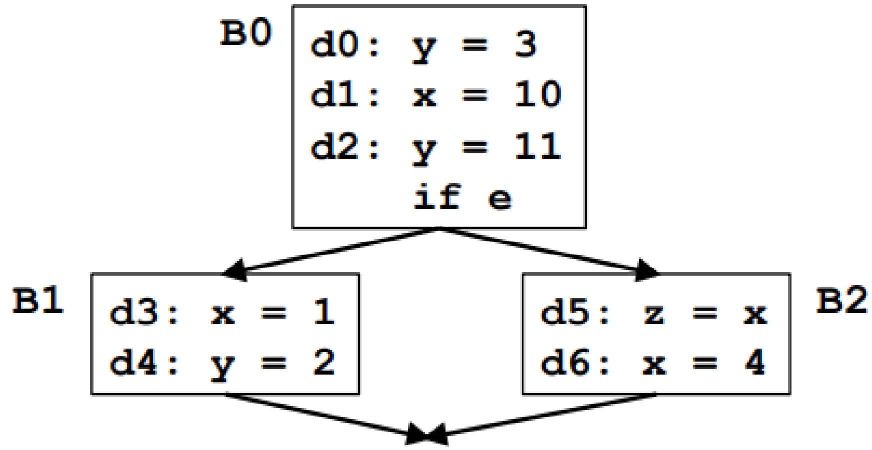


Static Program vs. Dynamic Execution



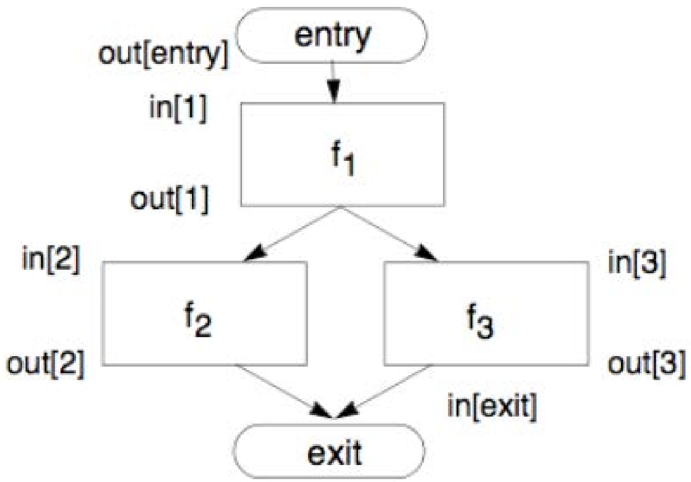
- Dynamically: Can have infinitely many possible execution paths
- Statically: Finite program
- Data flow analysis abstraction:
  – For each point in the program: combines information of all the instances of the same program point.
- Example of a data flow question:
  – Which definition defines the value used in statement "b = a"?

Reaching Definitions



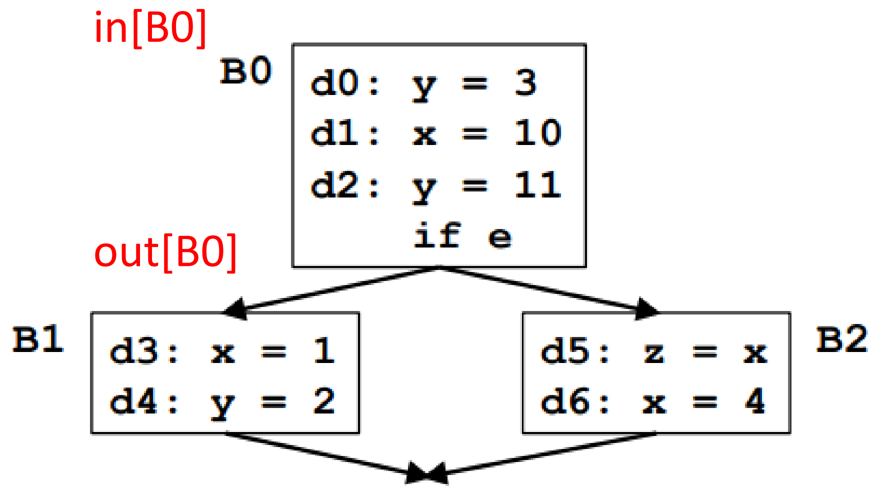
- Every assignment is a definition
- A definition d reaches a point p if there exists a path from the point immediately following d to p such that d is not killed (overwritten) along that path.
- Problem statement
  – For each point in the program, determine if each definition in the program reaches the point

Data Flow Analysis Schema



- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between in[b] and out[b] for all basic blocks b
  - Effect of code in basic block:
    - Transfer function fb relates in[b] and out[b], for same b
  - Effect of flow of control:
    - relates $out[b_1]$, $in[b_2]$ if b_1 and b_2 are adjacent
- Find a solution to the equations

Effects of a Statement

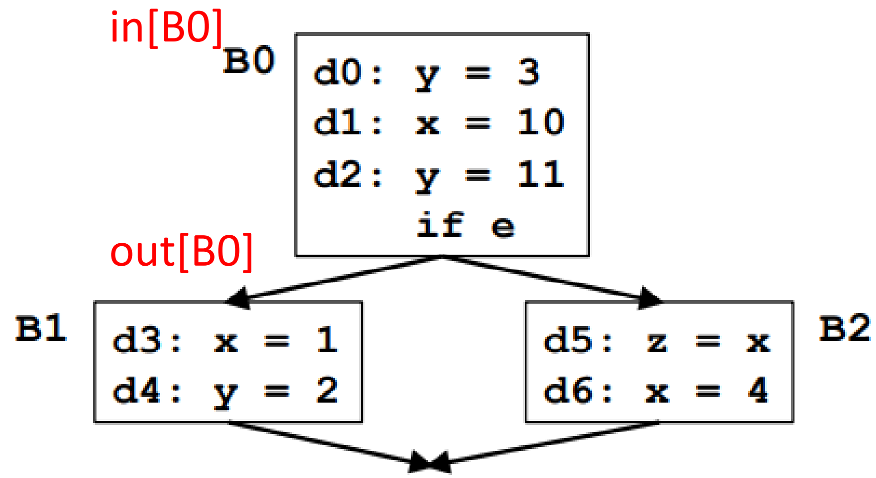


- f_s : A transfer function of a statement s
  - abstracts the execution with respect to the problem of interest
- For a statement s (d: x = y + z)

  $$out[s] = f_s(in[s]) = Gen[s] \cup (in[s]-Kill[s])$$

  - Gen[s]: definitions generated, Gen[s] = {d}
  - Kill[s]: set of all other defs to x in the rest of program
  - Propagated definitions: in[s] - Kill[s],

Effects of a Basic Block



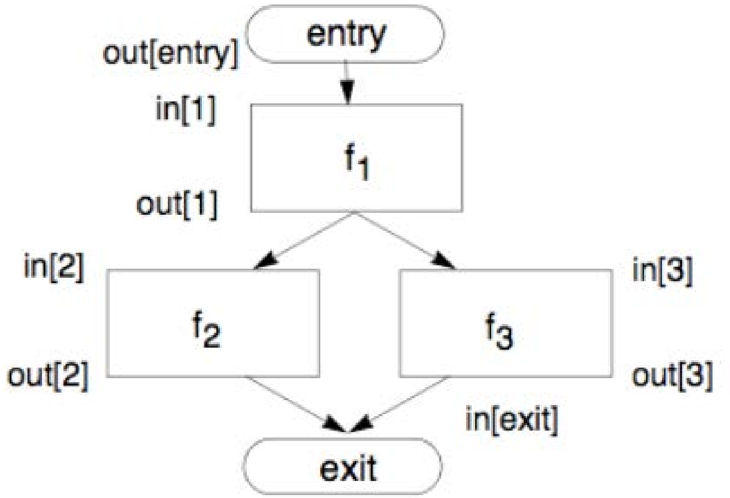
- f_s : A transfer function of a statement s

  out[s] = fs(in[s]) = Gen[s] U (in[s] - Kill[s])

- **Transfer function of a basic block B:** composition of transfer functions of statements in B $\mathbf{f_{B0}=f_{d0} \circ f_{d1} \circ f_{d2}}$

out[B] = f_B(in[B])
= f_{d1}(f_{d0}(in[B]))
= Gen[d1] U (Gen[d0] U (in[B] - Kill[d0])) - Kill[d1])
= (Gen[d1] U (Gen[d0] - Kill[d1])) U (in[B] - (Kill[d0] U Kill[d1]))
= Gen[B] U (in[B] - Kill[B])

Effects of the Edges (acyclic)

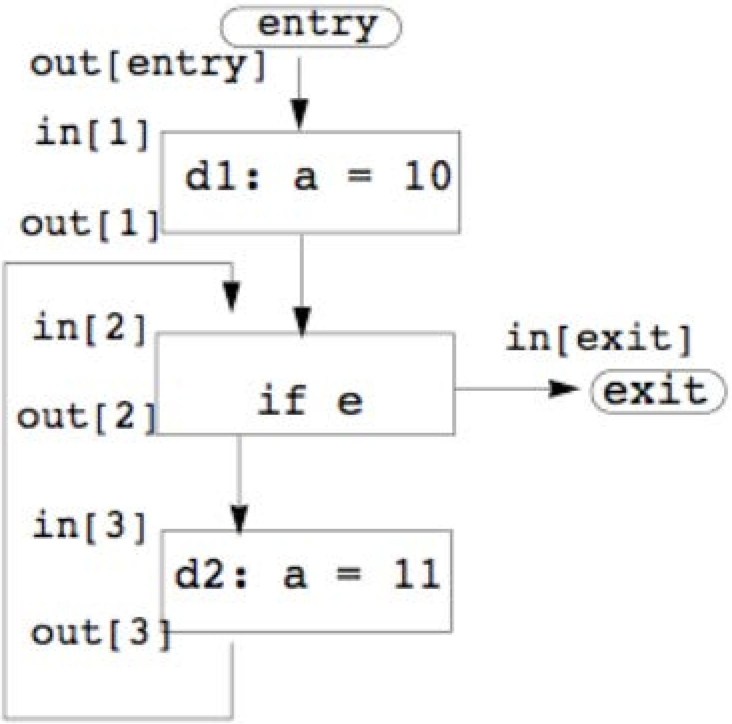


- Join node: a node with multiple predecessors
- meet operator ($\wedge$): U

$$in[b] = out[p_1] \cup out[p_2] \cup \ldots \cup out[p_n],$$

where $p_1, \ldots, p_n$ are all predecessors of b

Cyclic Graphs



- Equations still hold

  $out[b] = f_b(in[b])$

  $in[b] = out[p_1] \cup out[p_2] \cup \ldots \cup out[p_n]$,

  $p_1, \ldots, p_n$ are all predecessors of b
- Find: least fixed point solution

Reaching Definitions: Iterative Algorithm

input: control flow graph CFG = (N, E, Entry, Exit)

    out[Entry] = Ø   // Boundary condition
    For each basic block B other than Entry
        out[B] = Ø   // Initialization

    While (Changes to any out[] occur) {   // iterate
        For each basic block B other than Entry {
            in[B] = ∪ (out[p]), for all predecessors p of B
            out[B] = f_B(in[B])   // out[B]=gen[B] ∪ (in[B]-kill[B]) }

Data structure?

Bit-vector: one bit for each definition, X U Y= X or Y, X-Y = X and (not Y)

- Correctness
- Precision: how good is the answer?
- Convergence: will the analysis terminate?
- Complexity: how long does it take?

input: control flow graph CFG = (N, E, Entry, Exit)

out[Entry] = Ø // Boundary condition

For each basic block B other than Entry

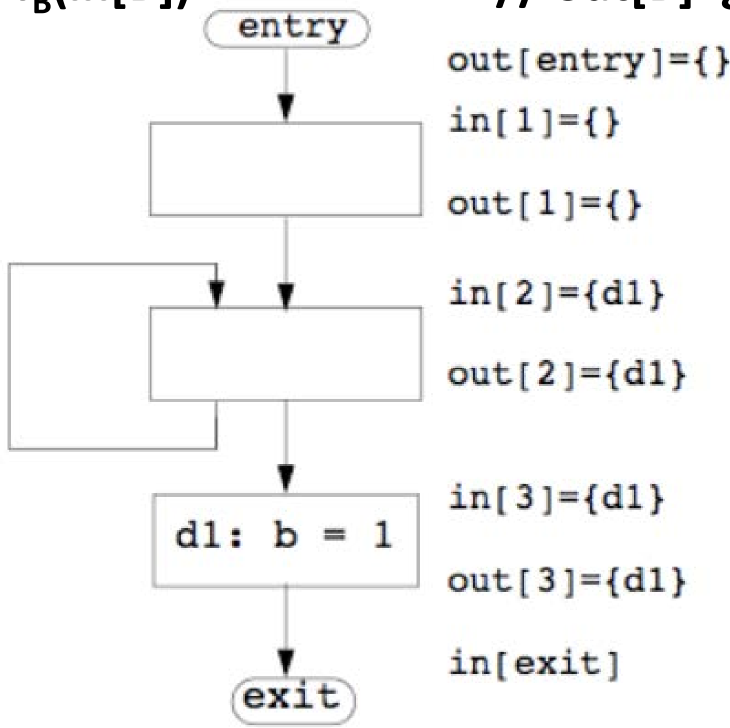
out[B] = Ø // Initialization

While (Changes to any out[] occur) { // iterate

For each basic block B other than Entry {

in[B] = ∪ (out[p]), for all predecessors p of B

out[B] = f_B(in[B]) // out[B]=gen[B]∪(in[B]-kill[B]) }



Will the worklist algorithm generate this answer?

Summary of Reaching Definitions

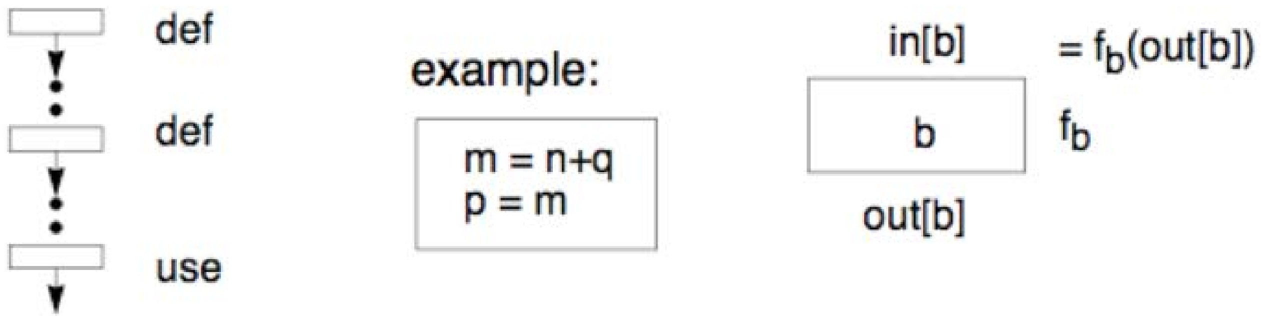
	Reaching Definitions
Domain	Sets of definitions
Transfer function f_b	forward: $out[b] = f_b(in[b])$ $f_b(x) = Gen_b \cup (x - Kill_b)$ Gen_b: definitions in b $Kill_b$: killed defs
Meet Operation	$\cup$
Boundary Condition	$out[entry] = \emptyset$
Initial interior points	$out[b] = \emptyset$

Liveness analysis

- Definition
  - A variable **v** is live at point **p** if the value of **v** is used along some path in the flow graph starting at **p**.
  - Otherwise, the variable is dead.
  - Dead code elimination in optimization
- Problem statement
  - **For each basic block**
    - **determine if each variable is live in each basic block**
  - **Size of bit vector: one bit for each variable**

Effects of a Basic Block (Transfer Function)

- Observation: Trace uses back to the definitions



- Direction: backward, in[b] = fb(out[b])
- Transfer function for statement s: x = y + z
  - generate live variables: Use[s] = {y, z}
  - propagate live variables: out[s] - Def[s], Def[s] = {x}

in[s] = Use[s] ∪ (out(s)-Def[s])

- Transfer function for basic block b:

in[b] = Use[b] ∪ (out(b)-Def[b])

Composition of transfer functions of statements: $f_{B0}=f_{d2} \circ f_{d1} \circ f_{d0}$

Effects of the Edges

- Meet operator (∧):

$$out[b] = in[s_1] \cup in[s_2] \cup \ldots \cup in[s_n],$$

  where $s_1, \ldots, s_n$ are all **successors** of b
- Boundary condition $in[Exit] = \emptyset$

- Equations still hold

  $in[b] = f_b(out[b])$

  $out[b] = in[s_1] \cup in[s_2] \cup \ldots \cup in[s_n],$

  $s_1, \ldots, s_n$ are all successors of b
- Find: least fixed point solution

Liveness : Iterative Algorithm

input: control flow graph CFG = (N, E, Entry, Exit)

in[Exit] = $\emptyset$ // Boundary condition

For each basic block B other than Entry

in[B] = $\emptyset$ // Initialization

While (Changes to any in[] occur) { // iterate

For each basic block B other than Entry {

out[B] = $\cup$ (in[p]), for all successors p of B

in[B] = f_B(out[B]) // in[B]=Use[B] $\cup$ (out[B]-Def[B])

- **one bit for each variable**
- Correctness
- Convergence: will the analysis terminate?
- Speed: how long does it take?

Dataflow-Analysis Direction

Forward analysis

– Start at the beginning of a function's CFG, work along the control edges (e.g., reaching definitions)

Backward analysis

– Start at the end of a function's CFG, work against the control edges (e.g., live variables)

Framework

	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Direction	forward: $out[b] = f_b(in[b])$ $in[b] = \wedge\ out[pred(b)]$	backward: $in[b] = f_b(out[b])$ $out[b] = \wedge\ in[succ(b)]$
Transfer function f_b	$f_b(x) = Gen_b \cup (x - Kill_b)$ Gen_b: definitions in b $Kill_b$: killed defs	$fb(x) = Use_b \cup (x - Def_b)$ Use_b: used in b Def_b: defined in b
Meet Operation	$\cup$	$\cup$
Boundary Condition	$out[entry] = \emptyset$	$in[exit] = \emptyset$
Initial interior points	$out[b] = \emptyset$	$in[b] = \emptyset$

Problem "Must-Reach" Definitions

- A definition **d** (a = b+c) must reach point **p** iff
  - **d** appears at least once along on all paths leading to **p**
  - **a** is not redefined along any path after last appearance of d and before p
- How do we formulate the data flow algorithm for this problem?

Framework

	Reaching Definitions	Must-Reach Definitions
Domain	Sets of definitions	
Direction	forward: $out[b] = f_b(in[b])$ $in[b] = \wedge\ out[pred(b)]$	
Transfer function f_b	$f_b(x) = Gen_b \cup (x - Kill_b)$ Gen_b: definitions in b $Kill_b$: killed defs	
Meet Operation	$\cup$	
Boundary Condition	$out[entry] = \emptyset$	
Initial interior points	$out[b] = \emptyset$	

Foundation of DataFlow Analysis

1. Semi-lattice (set of values, meet operator)
2. Transfer functions
3. Correctness, precision and convergence
4. Meaning of Data Flow Solution

Purpose of a Framework

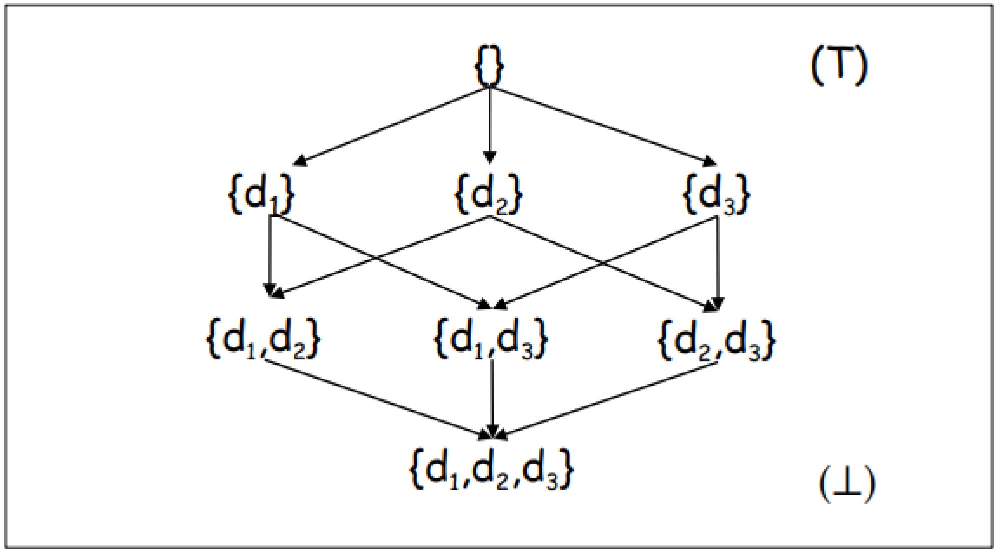
- Purpose 1
  – Prove properties of entire family of problems once and for all
    - Will the program converge?
    - What does the solution to the set of equations mean?
- Purpose 2:
  – Aid in software engineering: re-use code

The Data-Flow Framework

- Data-flow problems (F, V, ∧) are defined by
  - A (meet) semi-lattice （V, ∧ ）
    - domain of values V
    - meet operator ∧: V x V -> V (Greatest Lower Bound)
      - idempotent: $x \wedge x = x$
      - commutative: $x \wedge y = y \wedge x$
      - associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
  - A family of transfer functions F: V -> V

Example of a Semi-Lattice Diagram

- $(V, \wedge)$: $V = \{x \mid \text{such that } x \subseteq \{d_1,d_2,d_3\}\}$, $\wedge = \cup$



- Greatest lower bound:

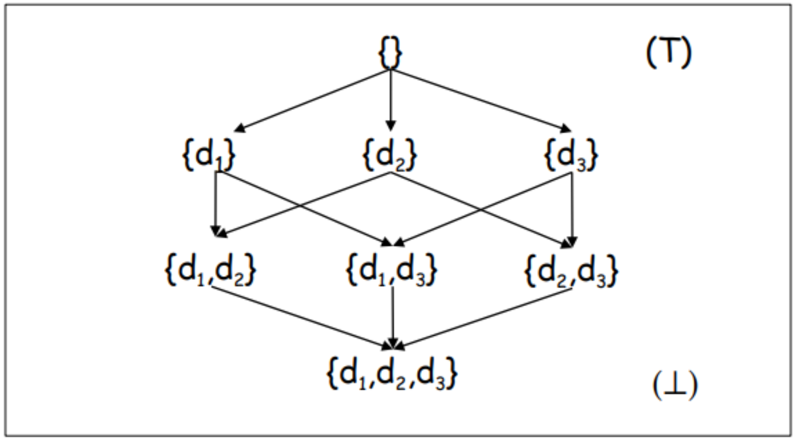
  $x \wedge y$ = first common descendant of x & y

- A meet semi-lattice is bounded if
  - there exists a top element T (i.e., {}), such that $x \wedge T = x$ for all x.
- A bottom element $\perp$ (i.e., $\{d_1,d_2,d_3\}$) exists, if

  $x \wedge \perp = \perp$ for all x.

Meet Semi-Lattices vs Partially Ordered Sets

A meet-semilattice is a partially ordered set which has a **meet** (or **greatest lower bound**) for any nonempty finite subset.



- Greatest lower bound:

  $x \wedge y$ = first common descendant of x & y

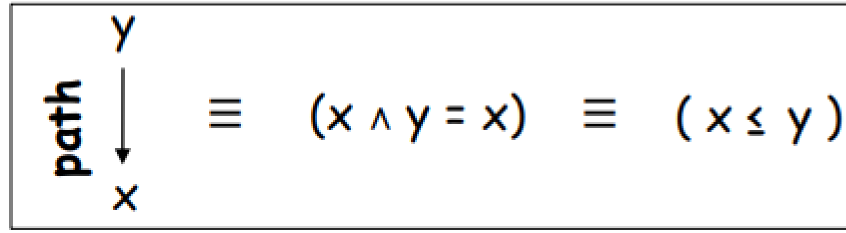
- Largest: top element T (i.e., {}), such that $x \wedge T = x$ for all x.
- Smallest: bottom element $\perp$ (i.e., $\{d_1,d_2,d_3\}$) exists, if

  $x \wedge \perp = \perp$ for all x.

A Meet Operator Defines a Partial Order

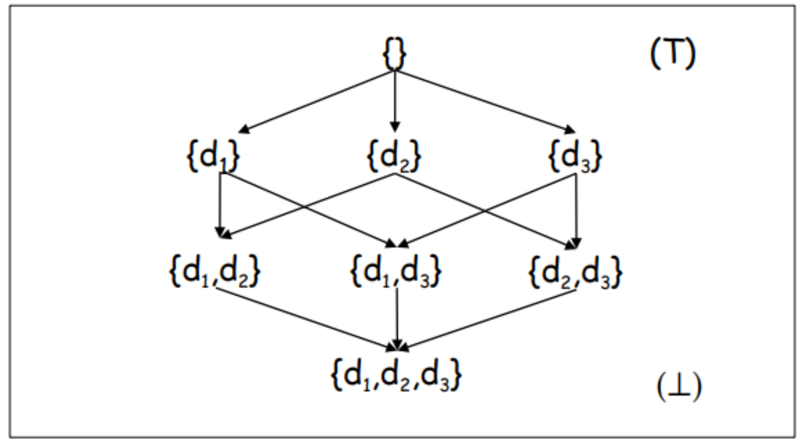
Partial order of a meet semi-lattice

$\leq$: $x \leq y$ if and only if $x \wedge y = x$



Meet operator: U

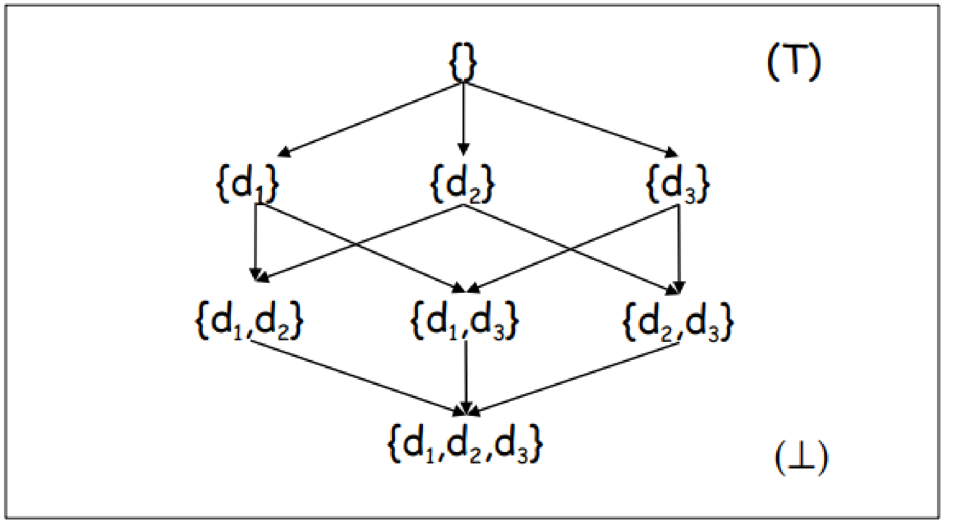
Partial order: ≤



- Properties of meet operator guarantee that $\leq$ is a partial order
  - Reflexive: $x \leq x$
  - Antisymmetric: if $x \leq y$ and $y \leq x$ then $x = y$
  - Transitive: if $x \leq y$ and $y \leq z$ then $x \leq z$

Drawing a Semi-Lattice Diagram

- $(x < y) \equiv (x \leq y) \wedge (x \neq y)$

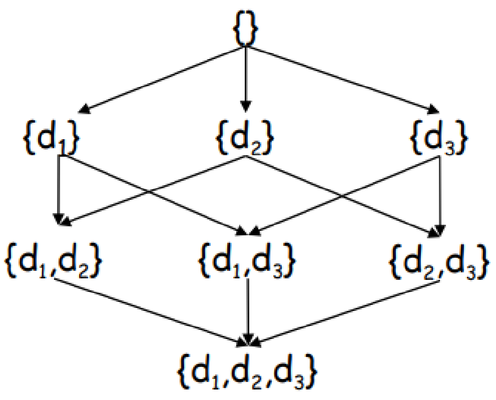


- A semi-lattice diagram:
  - Set of nodes: set of values
  - Set of edges $\{(y, x): x < y \text{ and } \neg\, \exists z \text{ s.t. } (x < z) \wedge (z < y)\}$

Summary

Three ways to define a semi-lattice:

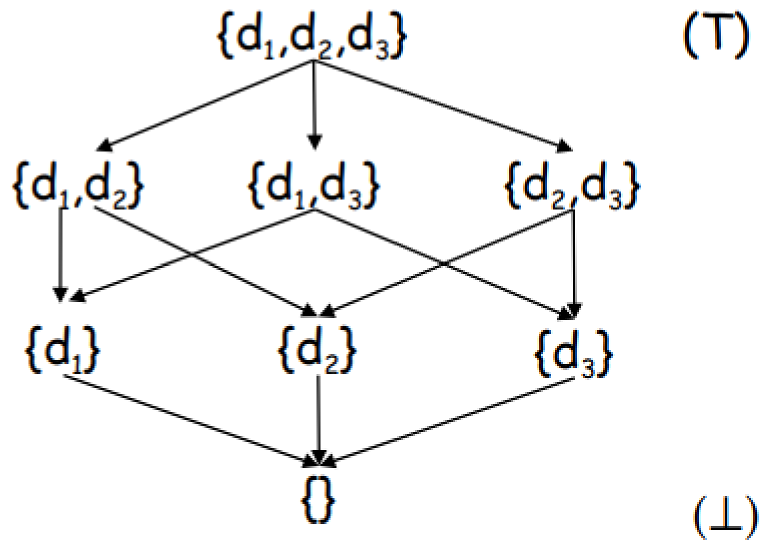
- Set V of values + meet operator $\wedge$: V x V -> V
  - idempotent: $x \wedge x = x$
  - commutative: $x \wedge y = y \wedge x$
  - associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- Set V of values + partial order with a greatest lower bound for any nonempty subset
  - Reflexive: $x \leq x$
  - Antisymmetric: if $x \leq y$ and $y \leq x$ then $x = y$
  - Transitive: if $x \leq y$ and $y \leq z$ then $x \leq z$
- A semi-lattice diagram



Another Example

Semi-lattice

- $V = \{x \mid \text{such that } x \subseteq \{d_1, d_2, d_3\}\}$
- $\wedge = \cap$
- $\leq$ is

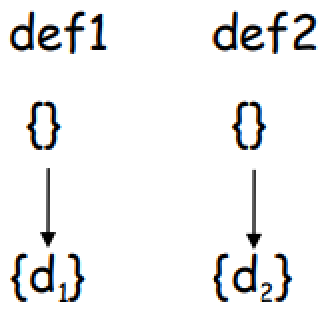


One Element at a Time

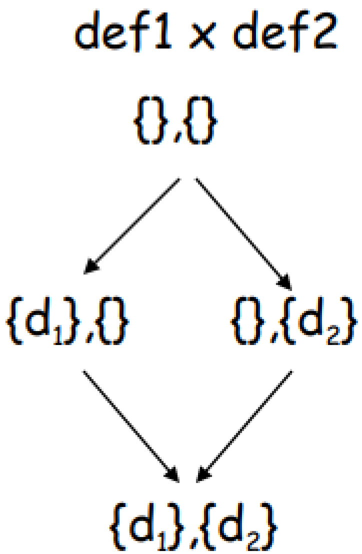
- A semi-lattice for data flow problems can get quite large:

  2^n elements for n live variables/reaching definitions
- A useful technique:
  - define semi-lattice for 1 element
  - product of semi-lattices for all elements

Example: Union of definitions



For each element

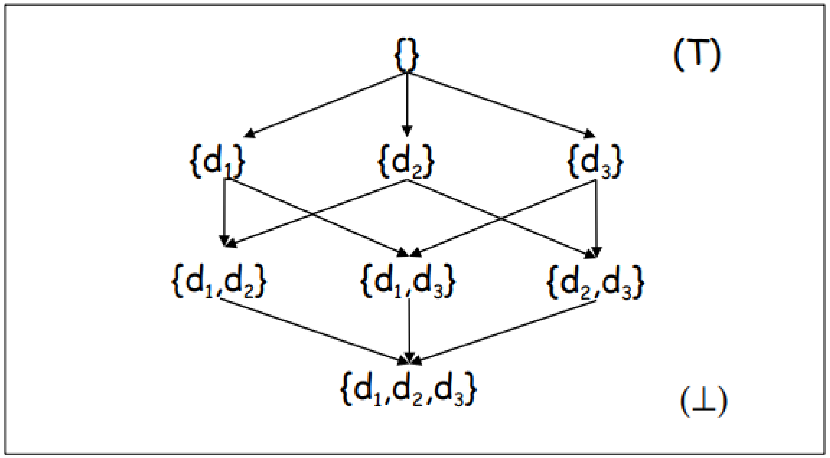


$\langle x_1,y_1\rangle \le \langle x_2,y_2\rangle$ iff $x_1 \le y_1$ and $x_2 \le y_2$

Descending Chain

- Definition : The height of a lattice is the largest number of > relations that will fit in a descending chain.

$$x_0 > x_1 > \ldots > x_n$$



- Height of values in reaching definitions?
- Important property: finite descending chains

The Data-Flow Framework

- Data-flow problems (F, V, $\wedge$) are defined by
  - A semi-lattice （ V, $\wedge$ ）
    - domain of values V
    - meet operator $\wedge$: V x V -> V
      - **idempotent: $x \wedge x = x$**
      - **commutative: $x \wedge y = y \wedge x$**
      - **associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$**
  - A family of transfer functions F: V -> V
    - Basic Properties f : V -> V
      - Has an identity function: $\exists f$ such that $f(x) = x$, for all x
      - Closed under composition: if $f_1, f_2 \in F$, $f_1 \circ f_2 \in F$

Monotonicity: 2 Equivalent Definitions

A framework (F, V, $\wedge$) is monotone iff

$$x \leq y \text{ implies } f(x) \leq f(y)$$

E.g.: Reaching definitions: $f(x) = \text{Gen} \cup (x - \text{Kill})$, $\wedge = \cup$

Let $x_1 \leq x_2$, $f(x_1)=\text{Gen} \cup (x_1 - \text{Kill}) \leq f(x_2)= \text{Gen} \cup (x_2 - \text{Kill})$

Equivalently, a framework (F, V, $\wedge$) is monotone iff

$$f(x \wedge y) \leq f(x) \wedge f(y),$$

E.g.: Reaching definitions: $f(x) = \text{Gen} \cup (x - \text{Kill})$, $\wedge = \cup$

$f(x_1 \wedge x_2) = (\text{Gen} \cup ((x_1 \cup x_2) - \text{Kill}))$

$f(x_1) \wedge f(x_2) = (\text{Gen} \cup (x_1 - \text{Kill})) \cup (\text{Gen} \cup (x_2 - \text{Kill}))$

$= \text{Gen} \cup (x_1\text{-Kill}) \cup (x_2\text{-Kill}) \leq$ (indeed =) $f(x_1 \wedge x_2)$

Distributivity

- A framework (F, V, $\wedge$) is distributive iff

$$f(x \wedge y) = f(x) \wedge f(y),$$

E.g.: Reaching definitions: $f(x) = Gen \cup (x - Kill)$, $\wedge = \cup$

$f(x_1 \wedge x_2) = (Gen \cup ((x_1 \cup x_2) - Kill))$

$f(x_1) \wedge f(x_2) = (Gen \cup (x_1 - Kill)) \cup (Gen \cup (x_2 - Kill))$

$= Gen \cup (x_1 - Kill) \cup (x_2 - Kill) = f(x_1 \wedge x_2)$

A special case of a monotone framework

Framework

	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Direction	forward: $out[b] = f_b(in[b])$ $in[b] = \wedge\ out[pred(b)]$	backward: $in[b] = fb(out[b])$ $out[b] = \wedge\ in[succ(b)]$
Transfer function f_b	$f_b(x) = Gen_b \cup (x - Kill_b)$ Gen_b: definitions in b $Kill_b$: killed defs	$fb(x) = Use_b \cup (x - Def_b)$ Use_b: used in b Def_b: defined in b
Meet Operation	$\cup$	$\cup$
Boundary Condition	$out[entry] = \emptyset$	$in[exit] = \emptyset$
Initial interior points	$out[b] = \emptyset$	$in[b] = \emptyset$

General Iterative Algorithm

input: control flow graph CFG = (N, E, Entry, Exit)

    out[Entry] = ∅ // Boundary condition

    For each basic block B other than Entry

        out[B] = ∅ // Initialization

    While (Changes to any out[] occur) { // iterate

        For each basic block B other than Entry {

            in[B] = ∪ (out[p]), for all predecessors p of B

            out[B] = f_B(in[B]) // out[B]=gen[B]∪(in[B]-kill[B]) }

input: control flow graph CFG = (N, E, Entry, Exit)

    out[Entry] = v_{entry} // Boundary condition

    For each basic block B other than Entry

        out[B] = T // Initialization maximum

    While (Changes to any out[] occur) { // iterate

        For each basic block B other than Entry {

            in[B] = ∧ (out[p]), for all predecessors p of B // multiple paths meet

            out[B] = f_B(in[B]) // transfer function

Forward dataflow problem

General Iterative Algorithm

input: control flow graph CFG = (N, E, Entry, Exit)

    in[Exit] = ∅   // Boundary condition

    For each basic block B other than Entry

        in[B] = ∅   // Initialization

    While (Changes to any in[] occur) {   // iterate

        For each basic block B other than Entry {

            out[B] = ∪ (in[p]), for all successors p of B

            in[B] = f_B(out[B])   // in[B]=Use[B] ∪ (out[B]-Def[B])

input: control flow graph CFG = (N, E, Entry, Exit)

    in[Entry] = $\mathbf{v_{entry}}$ // Boundary condition

    For each basic block B other than Entry

        int[B] = **T**   // Initialization maximum

    While (Changes to any in[] occur) {   // iterate

        For each basic block B other than Entry {

            out[B] = ∧ (in[p]), for all successors p of B   // multiple paths meet

            in[B] = f_B(out[B])   // transfer function

Backward dataflow problem

General Iterative Algorithm

- If the algorithm terminates, then
  the result is a solution of the dataflow problem
- If the framework is monotone, then
  the solution found is the maximum fixed point w.r.t. ($\leq$)
- If the semi-lattice of the frame work is monotone and finite descending chain, then
  the algorithm always terminates

monotone dataflow framework + finite descending chain

$\Downarrow$

maximum fixed point solution

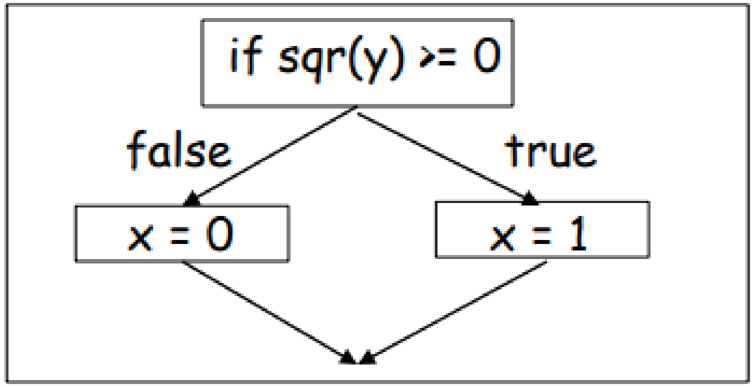
Behavior of iterative algorithm

- For each IN/OUT of an interior program point:
- **Invariant: new value ≤ old value in any step**
- Start with T (largest value)
- Proof by induction
  - 1st transfer function or meet operator: new value ≤ old value (T)
  - Meet operation:
    - Assume new inputs ≤ old inputs, then new output ≤ old output
  - Transfer function (in a monotone framework)
    - Assume new inputs ≤ old inputs, then new output ≤ old output
- Algorithm iterates until equations are satisfied
- Values do not come down unless some constraints drive them down.
- Therefore, finds the maximum solution that satisfies the equations

What Does the Solution Mean?

- IDEAL data flow solution
  - Let $f_1, \ldots, f_m \in F$, f_i is the transfer function for node i
    - f_p = composition of $f_{nk}, \ldots, f_{n1}$, for each path p =$n_1, \ldots, n_k$
    - f_p = identify function, if p is an empty path
  - For each node n: $\wedge f_{pi}$ (boundary value),

    for all possibly executed paths pi reaching n



Determining all possibly executed paths is undecidable

Meet-Over-Paths: MOP

- Meet-Over-Paths: MOP
  - Assume every edge is traversed
  - For each node n:
  - MOP(n) = $\wedge f_{pi}$ (boundary value), for all paths pi reaching n in CFG

- MOP VS. IDEAL
  - MOP includes more paths than IDEAL
  - MOP = IDEAL $\wedge$ Result(Unexecuted-Paths), MOP $\leq$ IDEAL
  - MOP is a "larger" solution, more conservative, safe

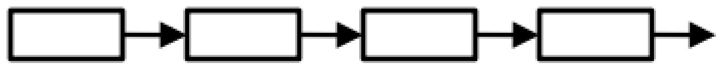
- MOP VS. maximum fixed point (MFP) MFP applies meet early
  - MFP $\leq$ MOP $\leq$ IDEAL
  - MFP, MOP are safe
  - If framework is distributive,
    MFP = MOP $\leq$ IDEAL

Summary

- A data flow framework
  - Semi-lattice
    - set of values (top)
    - meet operator
    - finite descending chains?
  - Transfer functions
    - summarizes each basic block
    - boundary conditions
- Properties of data flow framework:
  - Monotone framework and finite descending chains
    - $\Rightarrow$ iterative algorithm converges
    - $\Rightarrow$ finds maximum fixed point (MFP)
    - $\Rightarrow$ MFP $\leq$ MOP $\leq$ IDEAL
  - Distributive framework
    - $\Rightarrow$ MFP = MOP $\leq$ IDEAL

Efficiency of Iterative Data Flow

- Assume forward data flow for this discussion
- Speed of convergence depends on the ordering of nodes



input: control flow graph CFG = (N, E, Entry, Exit)

out[Entry] = $\mathbf{v}_{\mathbf{entry}}$

For each basic block B other than Entry

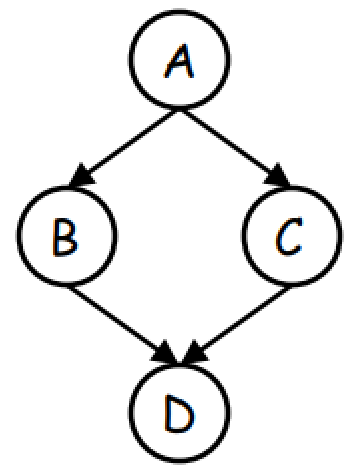
out[B] = $\top$

While (Changes to any out[] occur) {

For each basic block B other than Entry {

in[B] = $\wedge$ (out[p]), for all predecessors p of

out[B] = f_B(in[B])

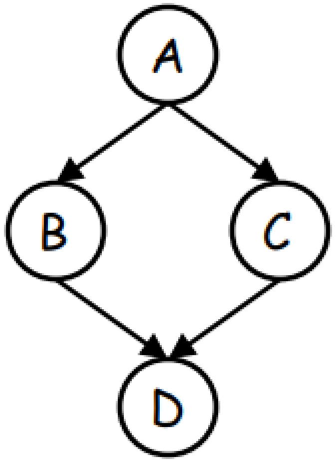


Forward dataflow problem

Better order? A,B,C,D or, D,B,C,A, ...

Reverse Postorder

- Preorder traversal: visit the parent before its children
- Postorder traversal: visit the children then the parent
- Preferred ordering: reverse postorder
  - depth first postorder visits the farthest node as early as possible
  - reverse postorder delays visiting farthest node



postorder traversals are: D B C A and D C B A
reverse postorder traversals are: A C B D and A B C D

“Reverse Post-Order” Iterative Data Flow

input: control flow graph CFG = (N, E, Entry, Exit)

    out[Entry] = $\mathbf{v_{entry}}$

    For each basic block B other than Entry

        out[B] = $\mathbf{T}$

        While (Changes to any out[] occur) {

        For each basic block B other than Entry **in reverse postorder** {

            in[B] = $\wedge$ (out[p]), for all predecessors p of

            out[B] = f_B(in[B])

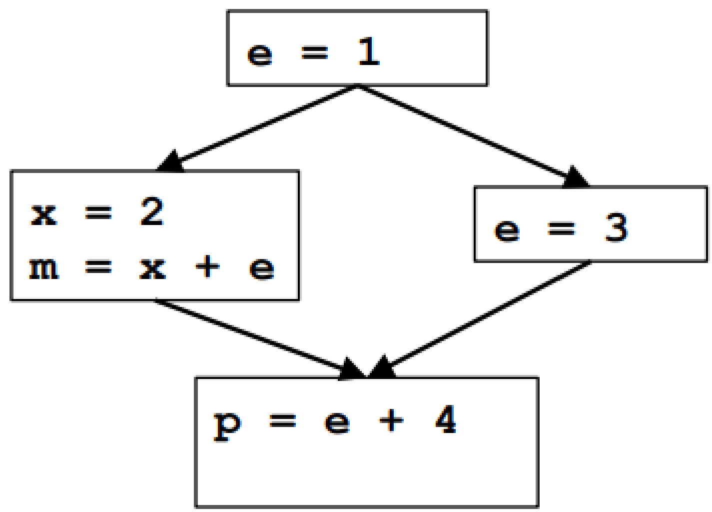
Forward dataflow problem

How to obtain reverse postorder traversal

1. One way is to run postorder traversal and push the nodes in a stack in postorder.
2. Then pop out the nodes to get the reverse postorder.

Constant Propagation/Folding

- At every basic block boundary, for each variable v
  - determine if v is a constant
  - if so, what is the value?
  - How do we know it is OK to globally propagate constants?
  - There are situations where it is incorrect:



  - To replace a use of x by a constant k we must know that:
    On every path to the use of x, the last assignment to x is x := k
    (Invariant #1)

Discussion

- The correctness condition is not trivial to check
- "All paths" includes paths around loops and through branches of conditionals
- Checking the condition requires global analysis
  - An analysis of the entire control-flow graph

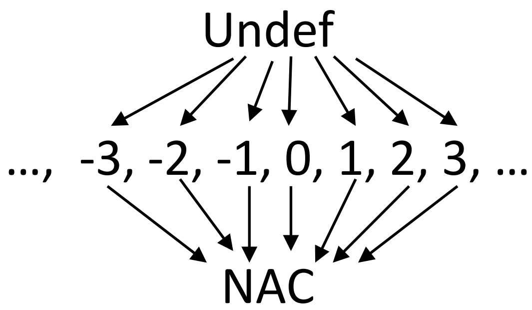
Can we use dataflow analysis?

- Semi-lattice
  - set of values (top): V ?
  - meet operator: $\wedge$?
  - finite descending chains?
- Transfer functions ?

Set of values

- To make the problem precise, we associate one of the following values with the variable x at every program point

Value	description
Undef	means that analysis hasn't determined if control reaches that point
c	constant c
NAC	is definitely not a constant 1. Assigned by an input value 2. Not a constant 3. Assigned different values via paths



- The set of values: product of semi-lattices, one component for each variable
- Represented by a map m: Var-> V

Meet

v_1	v_2	$v_1 \wedge v_2$
Undef	Undef	Undef
	c_2	c_2
	NAC	NAC
c_1	Undef	c_1
	c_2	NAC if c_1!=c_2 c_1 otherwise
	NAC	NAC
NAC	Undef	NAC
	c_2	NAC
	NAC	NAC

- Meet: $m_1 \wedge m_2 = m_3$, such that $m_1(x) \wedge m_2(x) = m_3(x)$
- i.e., $m_1 \leq m_2$ iff $m_1(x) \leq m_2(x)$ for all x in Var

Transfer functions

- Assume a basic block has only 1 instruction
- Non-assignment instruction: s

  f_s=identity function

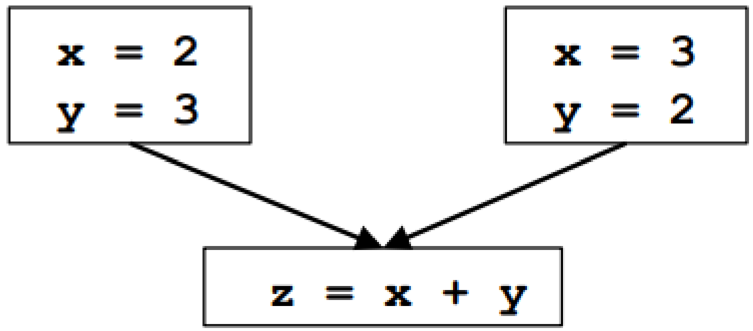
- Assignment s: x =e

  $f_s(m)=m'$

1. m(y)=m'(y), for all variable y!=x
2. m'(x) is defined as follows:
   - If e= c, then m'(x)=c
   - If e= y op z, m(y)=$c_1$and m(z)=c_2, then m(x)=c_1 op c_2
   - If e= y op z, and (m(y)=NAC or m(z)=NAC), then m'(x)=NAC
   - Otherwise, m'(x)=Undef
   - If e!=y op z (e.g., function call, assignment through a pointer), then m'(x)=NAC

- Use: $x \leq y$ implies $f(x) \leq f(y)$ to check if framework is monotone

Distributive?



- MFP<MOP
- Forward or backward?

Summary of Constant Propagation

- A useful optimization
- Illustrates:
  - abstract execution
  - an infinite semi-lattice
  - a non-distributive problem