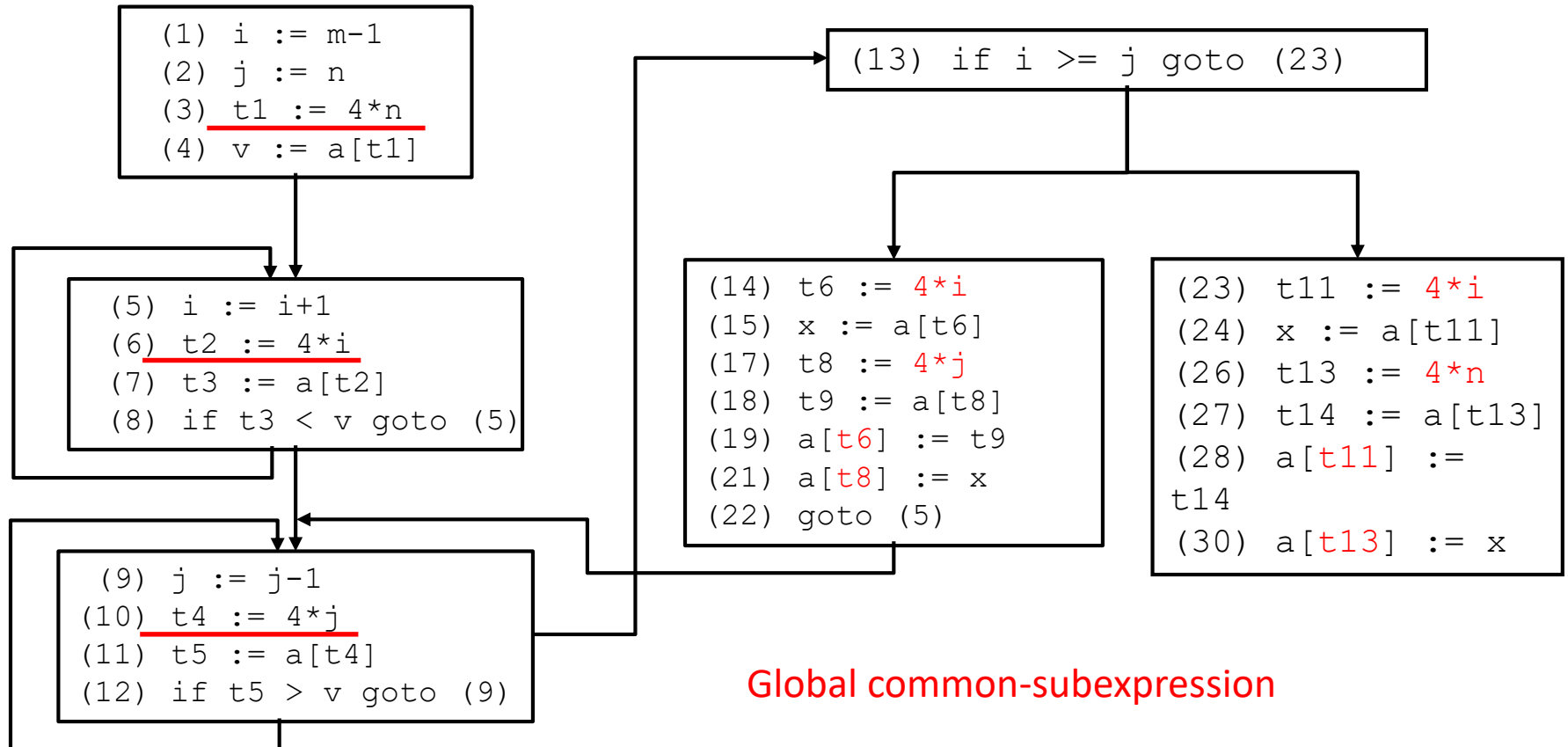


Dataflow Analysis

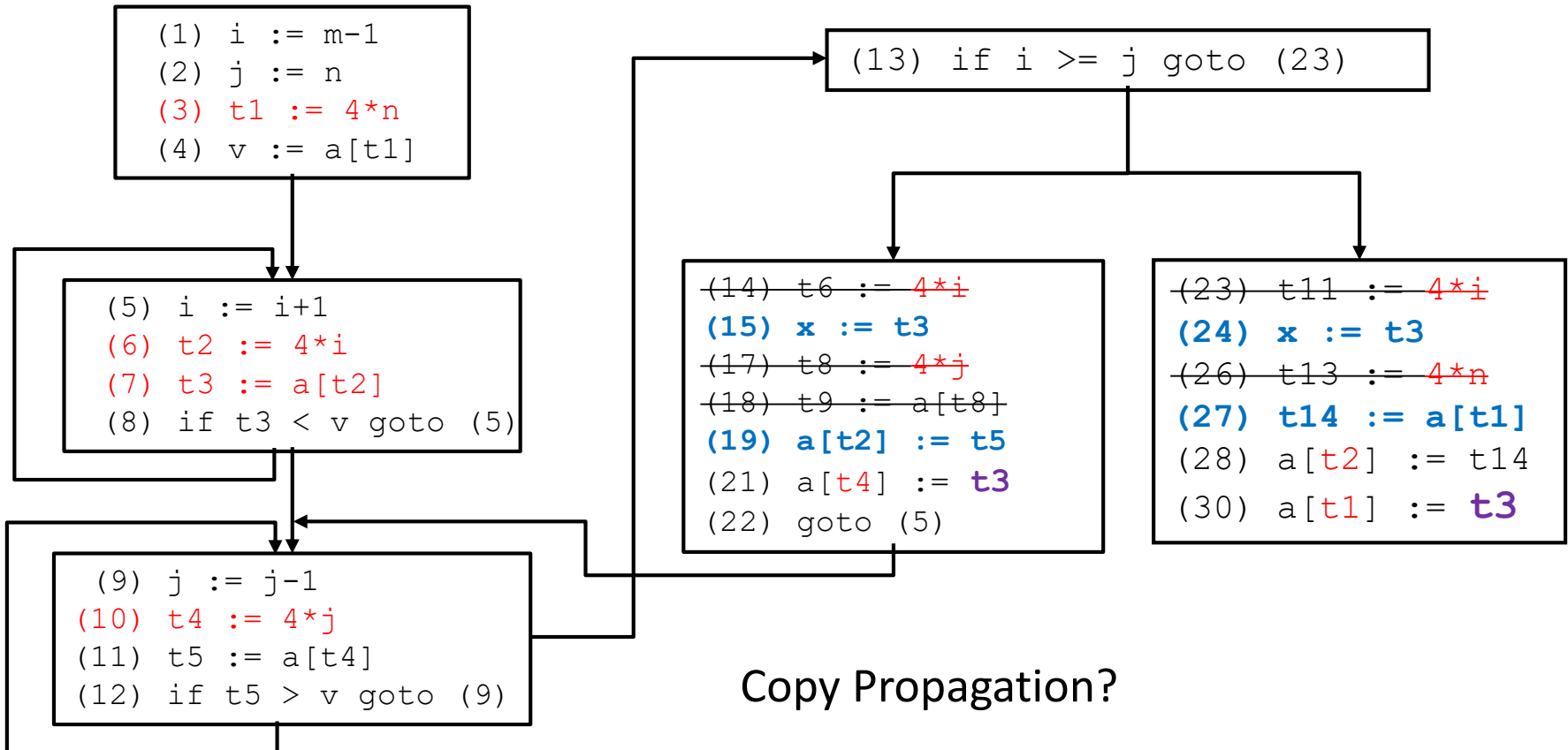
Local common-subexpression elimination and dead code elimination



Global common-subexpression

$i@14$ and 6 use the same definition?

Copy Propagation

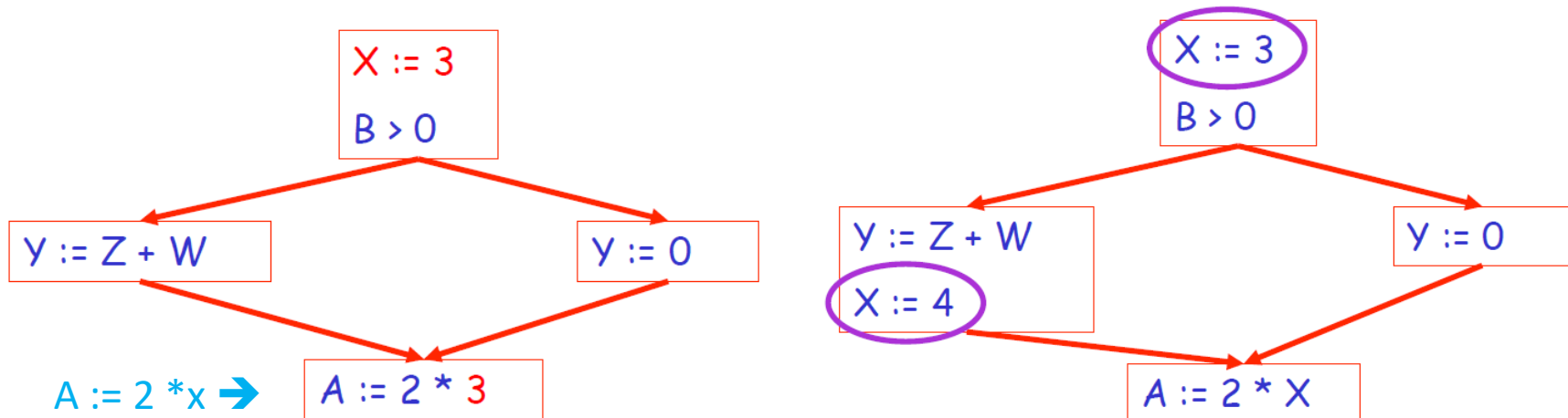


Copy Propagation?

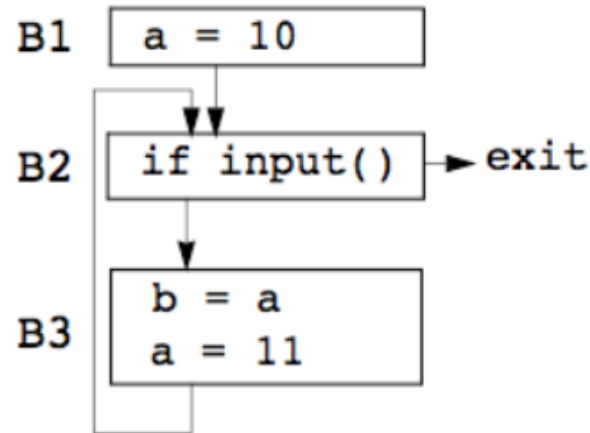
`x@21` uses the definition `@15`?

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

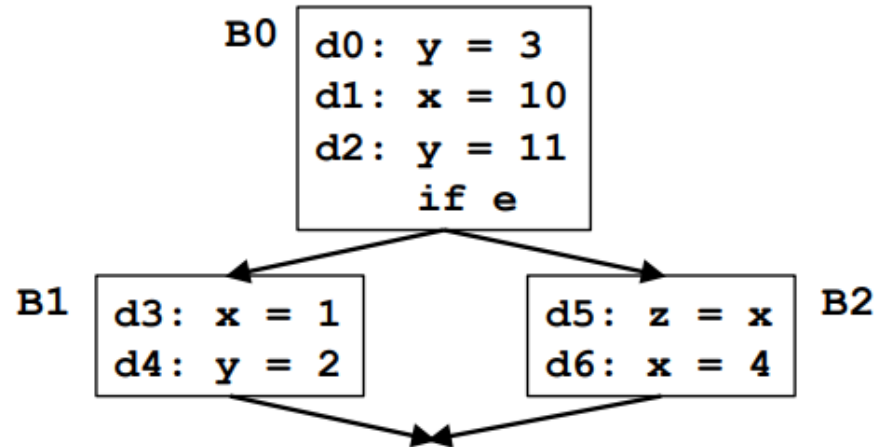


Static Program vs. Dynamic Execution



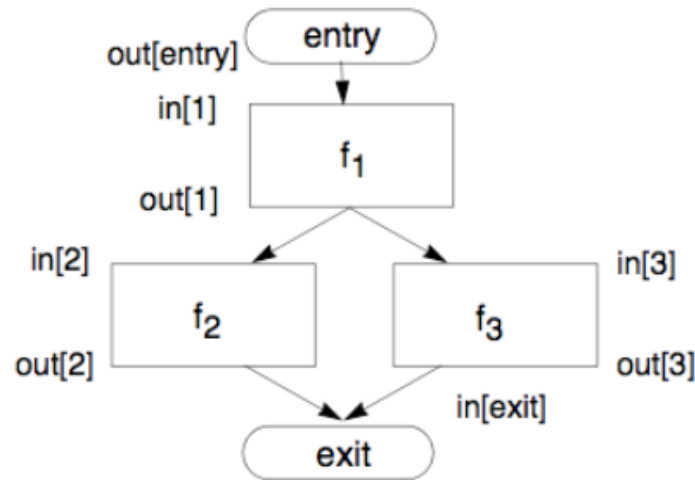
- **Dynamically**: Can have **infinitely** many possible execution paths
- **Statically**: Finite program
- Data flow analysis abstraction:
 - For each point in the program: combines information of all the instances of the same program point.
- Example of a data flow question:
 - Which definition defines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

Reaching Definitions



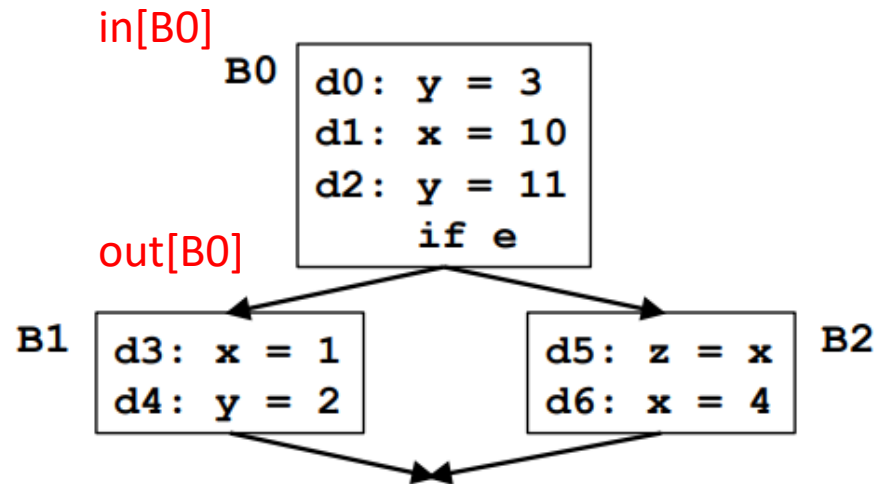
- Every assignment is a definition
- A **definition d reaches** a point **p** if there exists a path from the point immediately following **d** to **p** such that **d** is not killed (overwritten) along that path.
- **Problem statement**
 - For each point in the program, determine if each definition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

Data Flow Analysis Schema



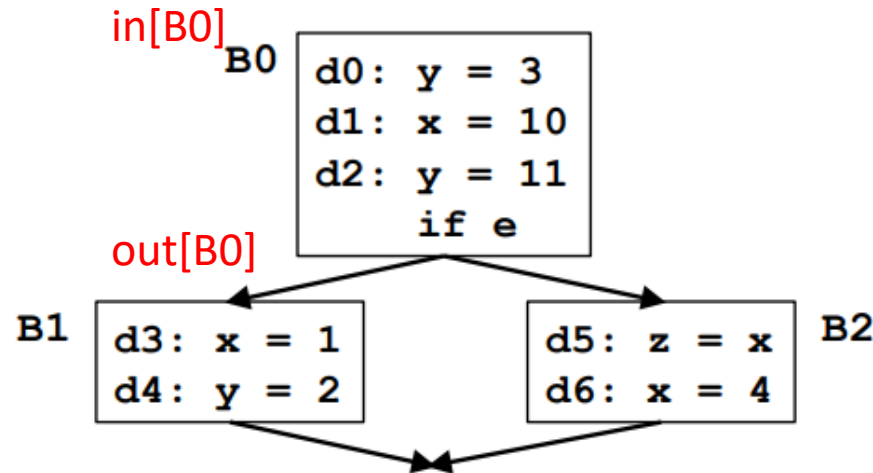
- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between $in[b]$ and $out[b]$ for all basic blocks b
 - Effect of code in basic block:
 - Transfer function fb relates $in[b]$ and $out[b]$, for same b
 - Effect of flow of control:
 - relates $out[b_1]$, $in[b_2]$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



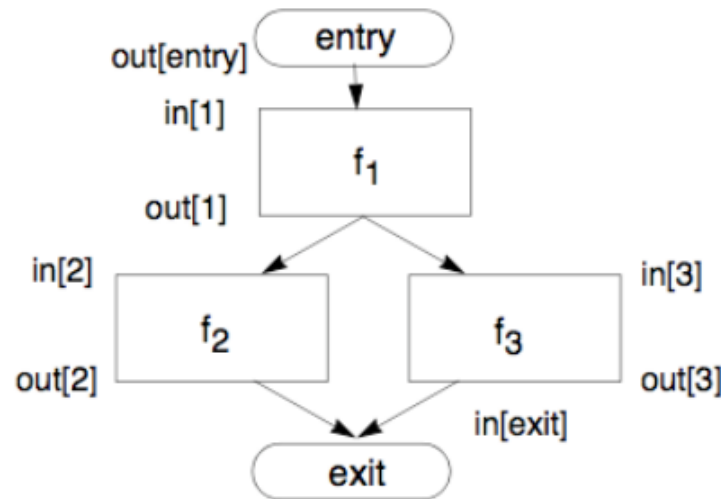
- f_s : A transfer function of a statement s

$$\text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s] - \text{Kill}[s])$$
- **Transfer function of a basic block B:** composition of transfer functions of statements in B

$$f_{B0} = f_{d2} \circ f_{d1} \circ f_{d0}$$

$$\begin{aligned}
 \text{out}[B] &= f_B(\text{in}[B]) \\
 &= f_{d1}(f_{d0}(\text{in}[B])) \\
 &= \text{Gen}[d1] \cup (\text{Gen}[d0] \cup (\text{in}[B] - \text{Kill}[d0]) - \text{Kill}[d1]) \\
 &= (\text{Gen}[d1] \cup (\text{Gen}[d0] - \text{Kill}[d1])) \cup (\text{in}[B] - (\text{Kill}[d0] \cup \text{Kill}[d1])) \\
 &= \text{Gen}[B] \cup (\text{in}[B] - \text{Kill}[B])
 \end{aligned}$$

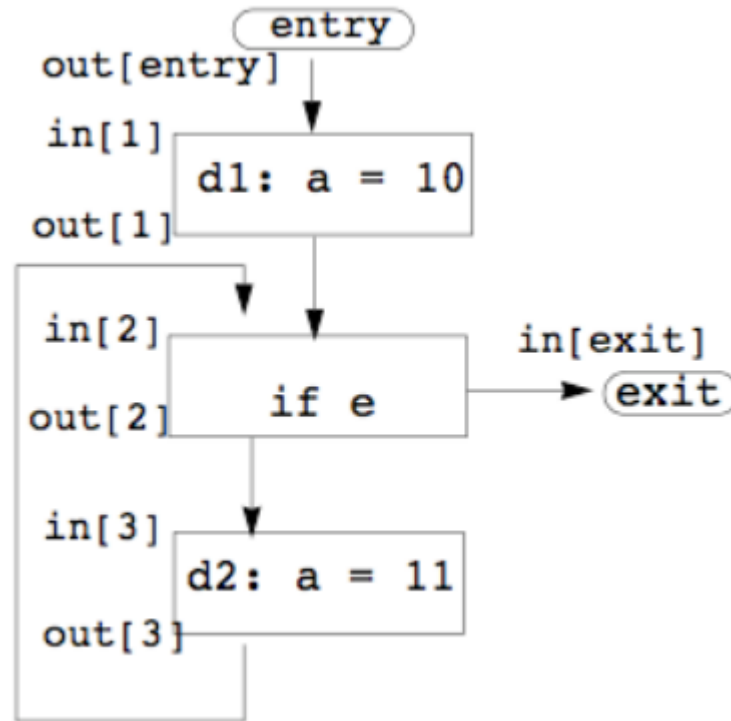
Effects of the Edges (acyclic)



- Join node: a node with multiple predecessors
- **meet operator (\wedge):** U
$$\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \dots \cup \text{out}[p_n],$$

where p_1, \dots, p_n are all predecessors of b

Cyclic Graphs



- Equations still hold
$$\text{out}[b] = f_b(\text{in}[b])$$
$$\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \dots \cup \text{out}[p_n],$$
$$p_1, \dots, p_n \text{ are all predecessors of } b$$
- Find: least fixed point solution

Reaching Definitions: Iterative Algorithm

input: control flow graph $CFG = (N, E, \text{Entry}, \text{Exit})$

$\text{out}[\text{Entry}] = \emptyset$ // Boundary condition

For each basic block B other than Entry

$\text{out}[B] = \emptyset$ // Initialization

While (Changes to any $\text{out}[]$ occur) { // iterate

For each basic block B other than Entry {

$\text{in}[B] = \bigcup (\text{out}[p])$, for all predecessors p of B

$\text{out}[B] = f_B(\text{in}[B])$ // $\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$ }

Data structure?

Bit-vector: one bit for each definition, $X \cup Y = X \text{ or } Y$, $X - Y = X \text{ and (not } Y)$

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

input: control flow graph $CFG = (N, E, \text{Entry}, \text{Exit})$

$\text{out}[\text{Entry}] = \emptyset$

// Boundary condition

For each basic block B other than Entry

$\text{out}[B] = \emptyset$

// Initialization

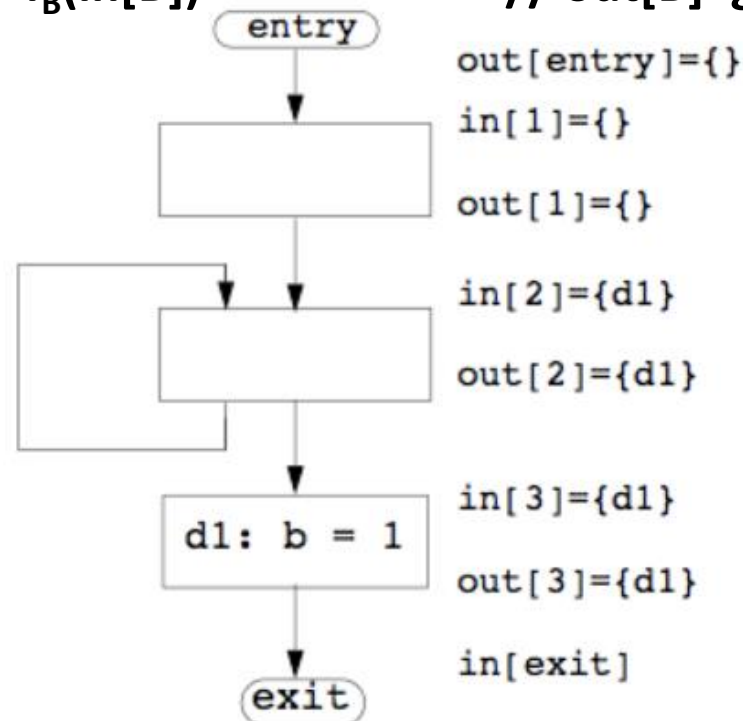
While (Changes to any $\text{out}[]$ occur) { // iterate

For each basic block B other than Entry {

$\text{in}[B] = \bigcup (\text{out}[p])$, for all predecessors p of B

$\text{out}[B] = f_B(\text{in}[B])$

// $\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$ }

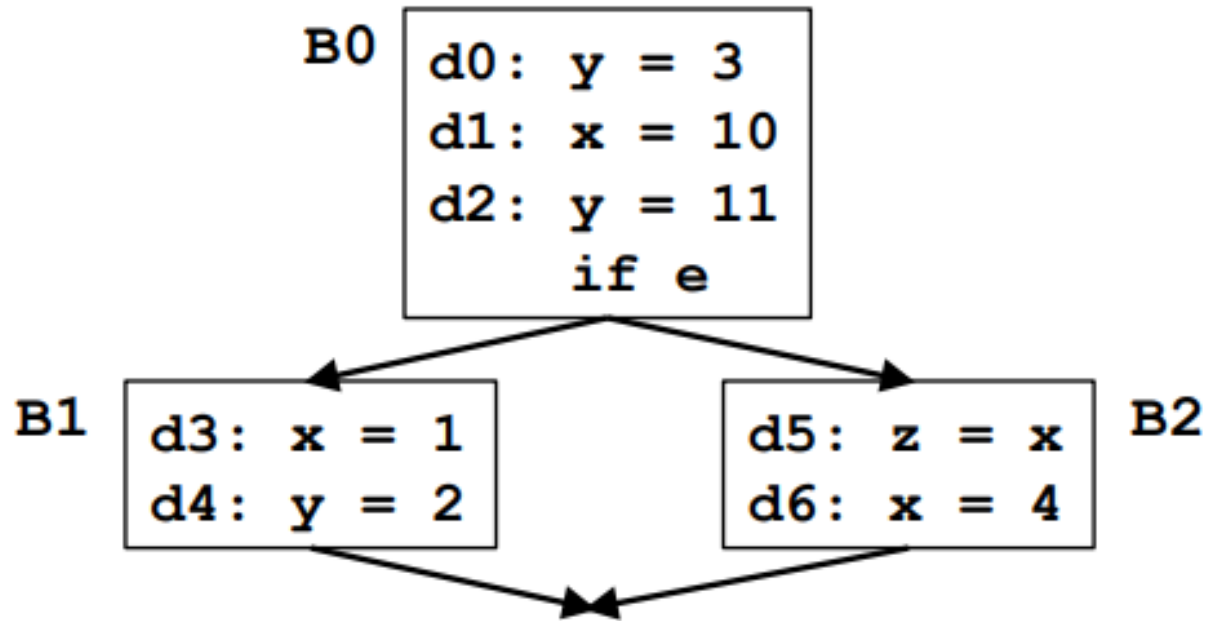


Will the worklist algorithm generate this answer?

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	$out[b] = \emptyset$

Exercise



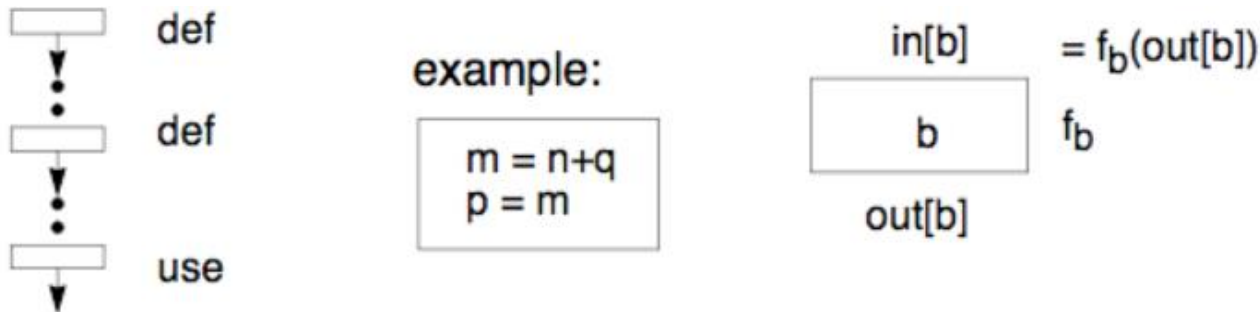
Compute Reaching Definitions

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**, $\text{in}[b] = f_b(\text{out}[b])$
- Transfer function for statement $s: x = y + z$
 - generate live variables: $\text{Use}[s] = \{y, z\}$
 - propagate live variables: $\text{out}[s] - \text{Def}[s], \text{Def}[s] = \{x\}$

$$\text{in}[s] = \text{Use}[s] \cup (\text{out}(s) - \text{Def}[s])$$

- Transfer function for basic block b :

$$\text{in}[b] = \text{Use}[b] \cup (\text{out}(b) - \text{Def}[b])$$

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

Effects of the Edges

- Meet operator (\wedge):

$$\text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup \dots \cup \text{in}[s_n],$$

where s_1, \dots, s_n are all **successors** of b

- Boundary condition $\text{in}[\text{Exit}] = \emptyset$
- Equations still hold
$$\text{in}[b] = f_b(\text{out}[b])$$
$$\text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup \dots \cup \text{in}[s_n],$$
$$s_1, \dots, s_n \text{ are all successors of } b$$
- Find: least fixed point solution

Liveness : Iterative Algorithm

input: control flow graph $CFG = (N, E, \text{Entry}, \text{Exit})$

$\text{in}[\text{Exit}] = \emptyset$ // Boundary condition

For each basic block B other than Entry

$\text{in}[B] = \emptyset$ // Initialization

While (Changes to any $\text{in}[]$ occur) { // iterate

For each basic block B other than Entry {

$\text{out}[B] = \bigcup (\text{in}[p])$, for all successors p of B

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

- **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)

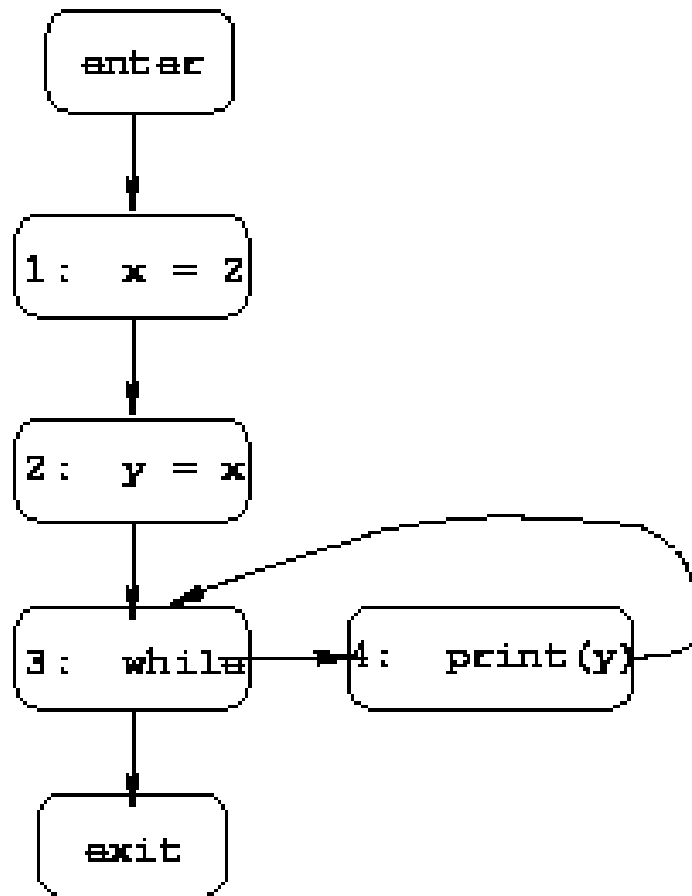
Backward analysis

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

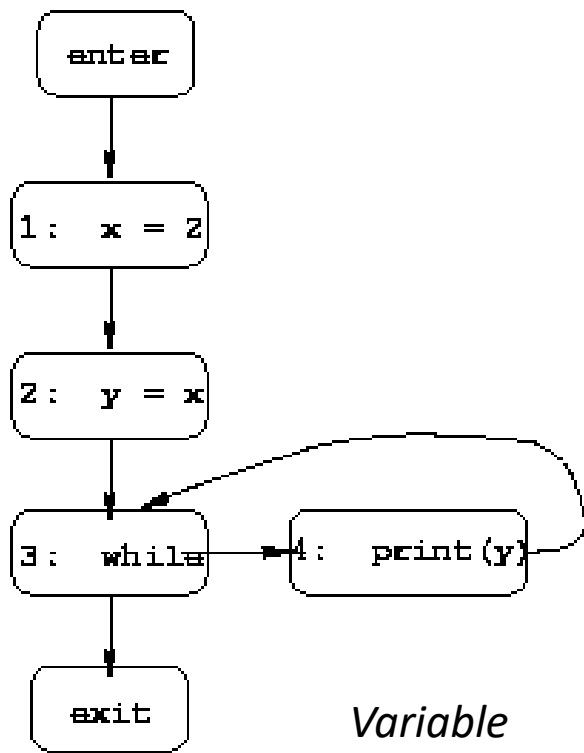
Framework

	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	$out[b] = \emptyset$	$in[b] = \emptyset$

Exercise



Write the equations for live-variable analysis, as well as the greatest and least solutions. Which is the desired solution, and why?



Variable	Least Solution	Greatest Solution
In[1]	{ }	{ }
Out[1]	{ x }	{ x }
In[2]	{ x }	{ x }
Out[2]	{ y }	{ x, y }
In[3]	{ y }	{ x, y }
Out[3]	{ y }	{ x, y }
In[4]	{ y }	{ x, y }
Out[4]	{ y }	{ x, y }

$out[enter] = in[1]$

$in[1] = out[1] - \{x\}$

$out[1] = in[2]$

$in[2] = (out[2] - \{y\}) \cup \{x\}$

$out[2] = in[3]$

$in[3] = out[3]$

$out[3] = in[exit] \cup in[4]$

$in[4] = out[4] \cup \{y\}$

$out[4] = in[3]$

$in[exit] = \emptyset$

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: $\text{out}[b] = f_b(\text{in}[b])$ $\text{in}[b] = \wedge \text{out}[\text{pred}(b)]$	
Transfer function f_b	$f_b(x) = \text{Gen}_b \cup (x - \text{Kill}_b)$ Gen_b : definitions in b Kill_b : killed defs	
Meet Operation	\cup	
Boundary Condition	$\text{out}[\text{entry}] = \emptyset$	
Initial interior points	$\text{out}[b] = \emptyset$	

Framework

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

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

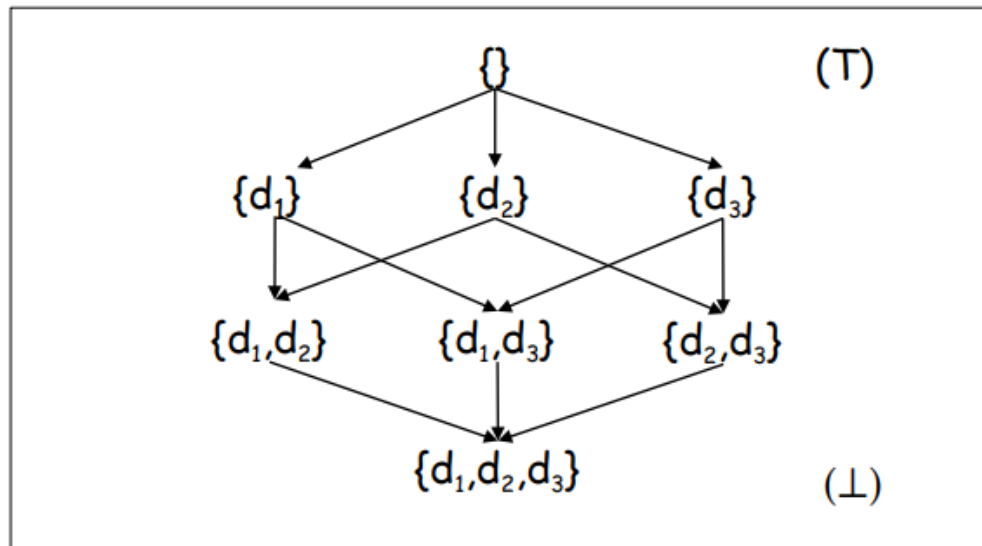
- Purpose 1
 - Prove properties of entire family of problems once and for all
 - Will the program converge?
 - What does the solution to the set of equations mean?
- Purpose 2:
 - Aid in software engineering: re-use code

The Data-Flow Framework

- Data-flow problems (F, V, \wedge) are defined by
 - A (meet) **semi-lattice** (V, \wedge)
 - domain of values **V**
 - meet operator **\wedge** : $V \times V \rightarrow V$ (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 **$F: V \rightarrow V$**

Example of a Semi-Lattice Diagram

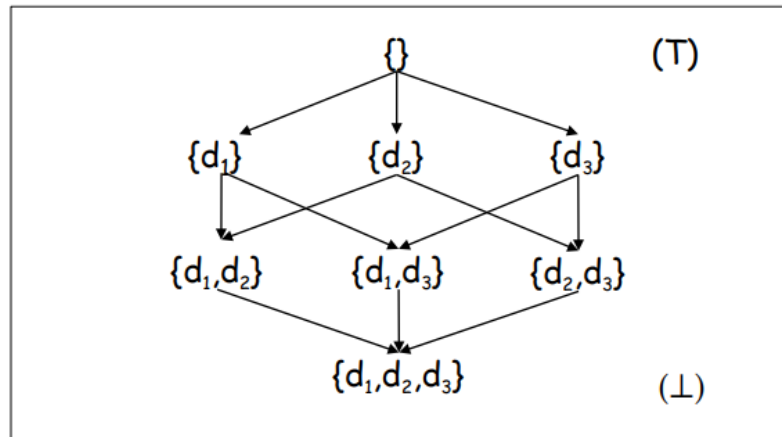
- $(V, \wedge) : V = \{x \mid \text{such that } x \subseteq \{d_1, d_2, d_3\}\}, \wedge = \cup$



- **Greatest lower bound:**
 $x \wedge y = \text{first common descendant of } x \text{ \& } y$
- 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
$$x \wedge \perp = \perp \text{ for all } x.$$

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:**

$x \wedge y = \text{first common descendant of } x \text{ \& } y$

- **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
$$x \wedge \perp = \perp \text{ for all } x.$$

A Meet Operator Defines a Partial Order

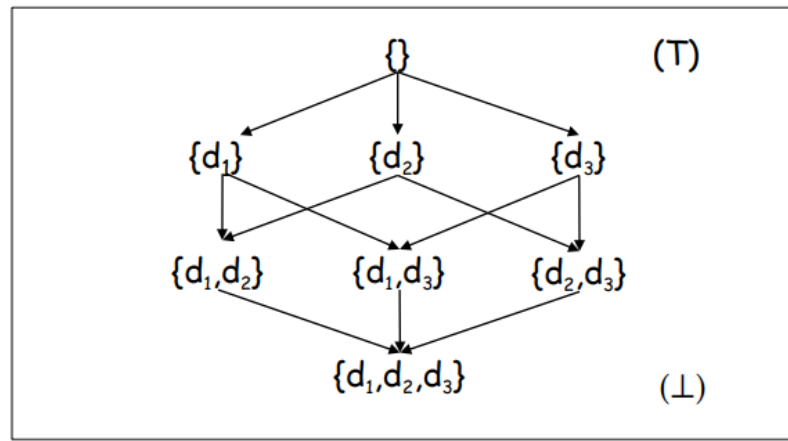
Partial order of a meet semi-lattice

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

$$\begin{array}{c} y \\ \text{path} \downarrow \\ x \end{array} \equiv (x \wedge y = x) \equiv (x \leq y)$$

Meet operator: \cup

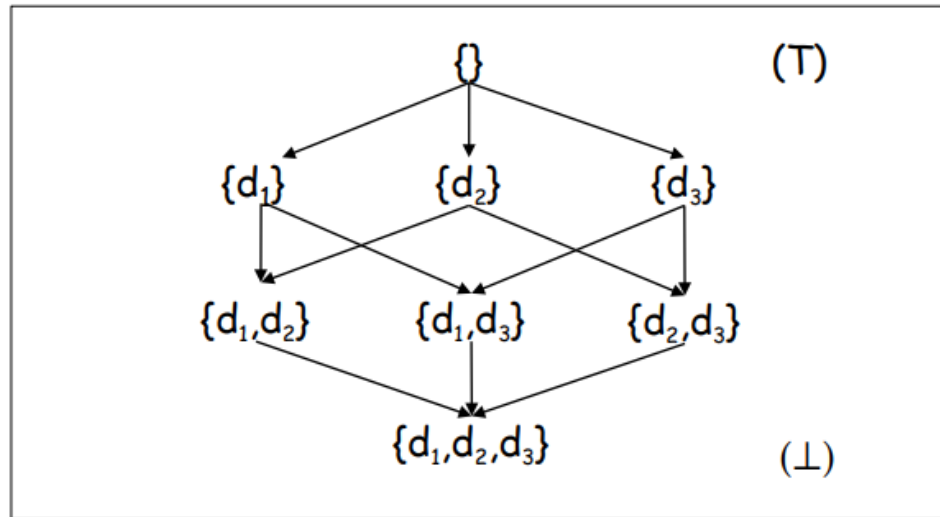
Partial order: \leq



- 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

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

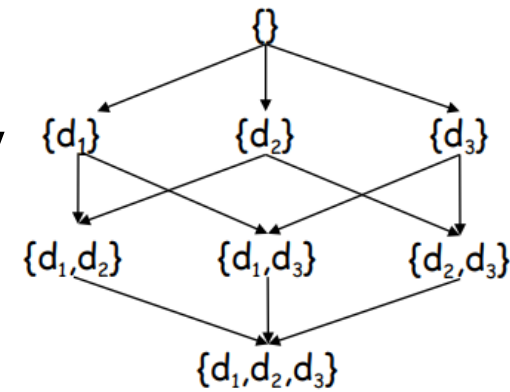


- A semi-lattice diagram:
 - Set of nodes: set of values
 - Set of edges $\{(y, x): x < y \text{ and } \neg \exists z \text{ s.t. } (x < z) \wedge (z < y)\}$

Summary

Three ways to define a semi-lattice:

- Set V of values + meet operator $\wedge: V \times V \rightarrow V$
 - 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

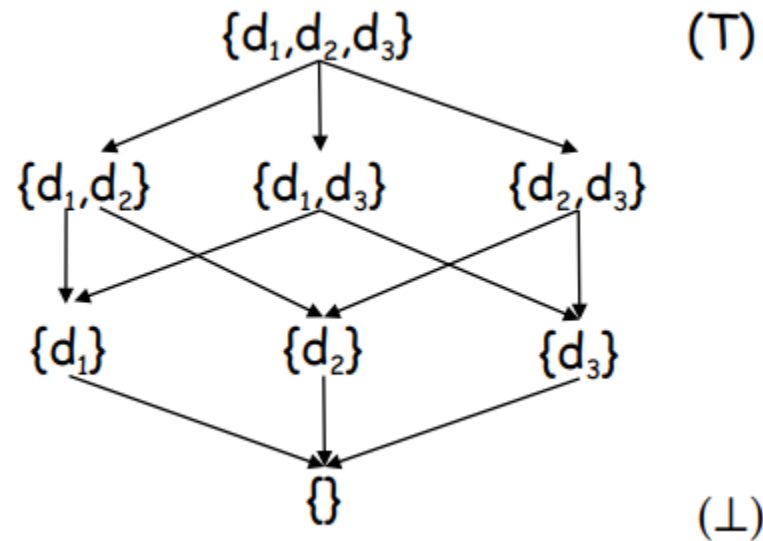


Another Example

Semi-lattice

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

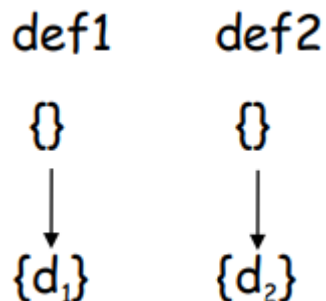
– \leq is



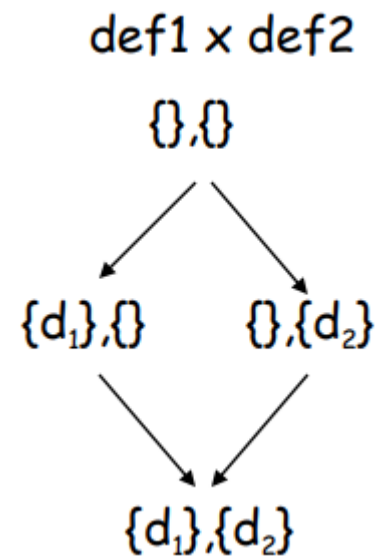
One Element at a Time

- 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

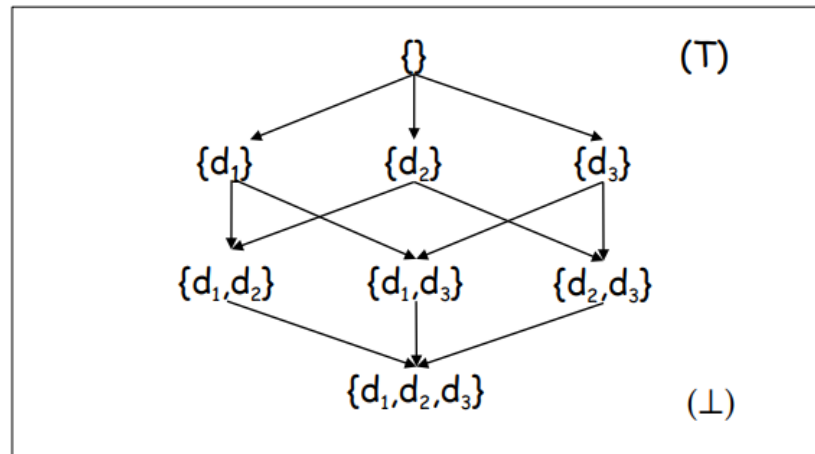


$\langle x_1, y_1 \rangle \leq \langle x_2, y_2 \rangle$ iff $x_1 \leq y_1$ and $x_2 \leq y_2$

Descending Chain

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

$$x_0 > x_1 > \dots > x_n$$



- 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 **$\wedge: V \times V \rightarrow V$**
 - **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 **$F: V \rightarrow V$**
 - Basic Properties $f: V \rightarrow V$
 - 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: $f(x) = \text{Gen} \cup (x - \text{Kill})$, $\wedge = \cup$

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

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

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

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

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

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

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

Distributivity

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

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

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

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

$$\begin{aligned} f(x_1) \wedge f(x_2) &= (\text{Gen } U (x_1 - \text{Kill})) \cup (\text{Gen } U (x_2 - \text{Kill})) \\ &= \text{Gen } U (x_1 - \text{Kill}) \cup (x_2 - \text{Kill}) = f(x_1 \wedge x_2) \end{aligned}$$

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	$out[b] = \emptyset$	$in[b] = \emptyset$

Review

	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	$out[b] = \emptyset$	$in[b] = \emptyset$

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

General Iterative Algorithm

input: control flow graph $CFG = (N, E, \text{Entry}, \text{Exit})$

out[Entry] = \emptyset // Boundary condition

For each basic block B other than Entry

out[B] = \emptyset // Initialization

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

For each basic block B other than Entry {

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

out[B] = $f_B(\text{in}[B])$ // out[B] = $\text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$ }

input: control flow graph $CFG = (N, E, \text{Entry}, \text{Exit})$

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

out[B] = $f_B(\text{in}[B])$ // transfer function

Forward dataflow problem

General Iterative Algorithm

input: control flow graph $CFG = (N, E, \text{Entry}, \text{Exit})$

$\text{in}[\text{Exit}] = \emptyset$ // Boundary condition

For each basic block B other than Entry

$\text{in}[B] = \emptyset$ // Initialization

While (Changes to any $\text{in}[]$ occur) { // iterate

For each basic block B other than Entry {

$\text{out}[B] = \bigcup (\text{in}[p])$, for all successors p of B

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

input: control flow graph $CFG = (N, E, \text{Entry}, \text{Exit})$

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

For each basic block B other than Entry

$\text{int}[B] = T$ // Initialization maximum

While (Changes to any $\text{in}[]$ occur) { // iterate

For each basic block B other than Entry {

$\text{out}[B] = \bigwedge (\text{in}[p])$, for all successors p of B // multiple paths meet

$\text{in}[B] = f_B(\text{out}[B])$ // 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



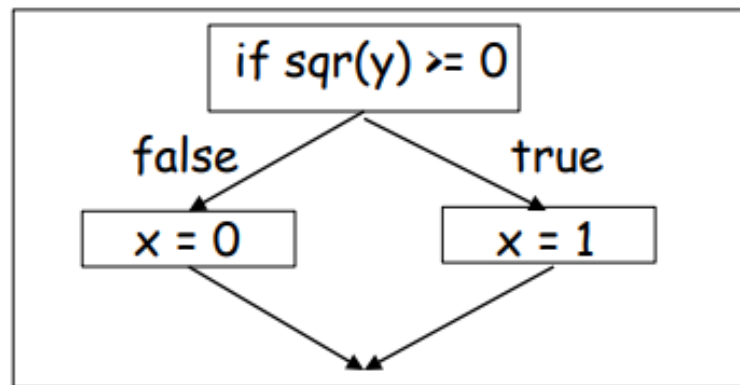
maximum fixed point solution

Behavior of iterative algorithm

- For each IN/OUT of an interior program point:
- **Invariant: new value \leq old value in any step**
- Start with T (largest value)
- Proof by induction
 - 1st transfer function or meet operator: new value \leq old value (T)
 - Meet operation:
 - Assume new inputs \leq old inputs, then new output \leq old output
 - Transfer function (in a monotone framework)
 - Assume new inputs \leq old inputs, then new output \leq 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 $f_1, \dots, f_m \in F$, f_i is the transfer function for node i
 - f_p = composition of f_{n_k}, \dots, f_{n_1} , for each path $p = n_1, \dots, n_k$
 - f_p = identity function, if p is an empty path
 - For each node n : $\bigwedge f_{p_i}$ (boundary value),
for all **possibly executed paths** p_i reaching n



Determining all possibly executed paths is undecidable

Meet-Over-Paths: MOP

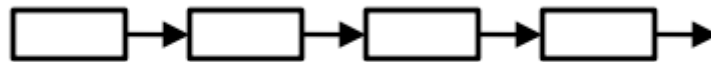
- Meet-Over-Paths: MOP
 - Assume every edge is traversed
 - For each node n :
 - $MOP(n) = \wedge f_{p_i}$ (boundary value), **for all paths p_i** reaching n in CFG
- MOP VS. IDEAL
 - MOP includes more paths than IDEAL
 - $MOP = IDEAL \wedge \text{Result}(\text{Unexecuted-Paths})$, $MOP \leq IDEAL$
 - 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)
 - $\Rightarrow \text{MFP} \leq \text{MOP} \leq \text{IDEAL}$
 - Distributive framework
 - $\Rightarrow \text{MFP} = \text{MOP} \leq \text{IDEAL}$

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 $CFG = (N, E, \text{Entry}, \text{Exit})$

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

For each basic block B other than Entry

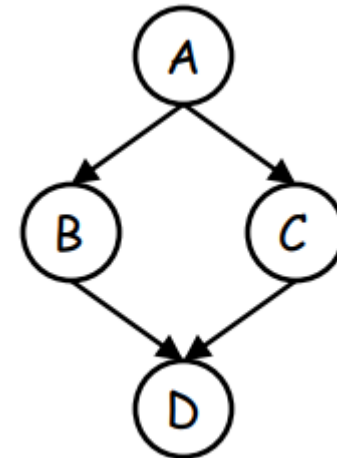
$\text{out}[B] = \mathbf{T}$

While (Changes to any $\text{out}[]$ occur) {

For each basic block B other than Entry {

$\text{in}[B] = \bigwedge (\text{out}[p])$, for all predecessors p of

$\text{out}[B] = f_B(\text{in}[B])$

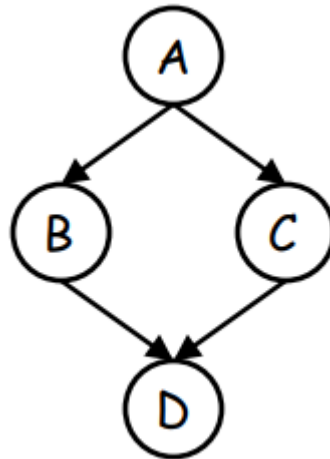


Forward dataflow problem

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 $CFG = (N, E, \text{Entry}, \text{Exit})$

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

For each basic block B other than Entry

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

While (Changes to any $\text{out}[]$ occur) {

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

$\text{in}[B] = \mathbf{\bigwedge} (\text{out}[p])$, for all predecessors p of

$\text{out}[B] = f_B(\text{in}[B])$

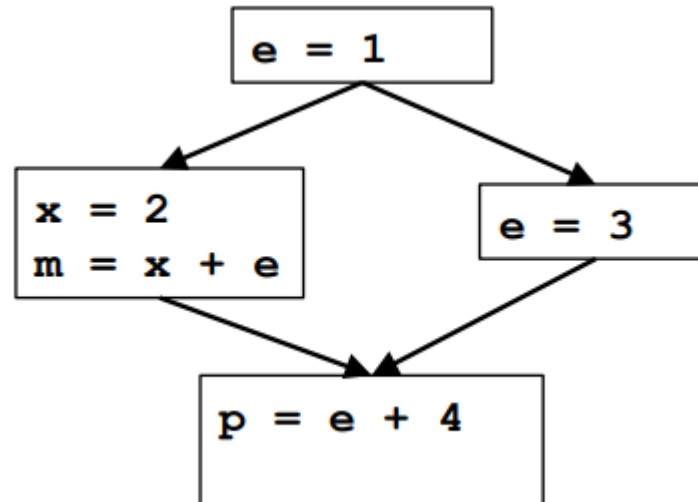
Forward dataflow problem

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

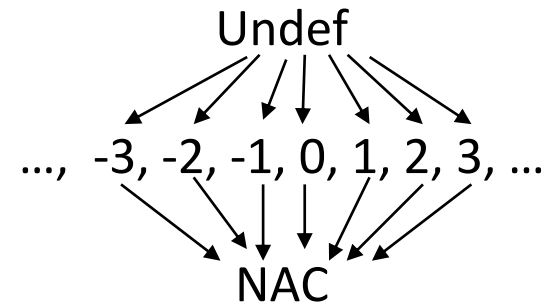
Can we use dataflow analysis?

- 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 <ol style="list-style-type: none">1. Assigned by an input value2. Not a constant3. Assigned different values via paths



- The set of values: product of semi-lattices,
one component for each variable
- Represented by a map $m: \text{Var} \rightarrow V$

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 $c_1 \neq c_2$ 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

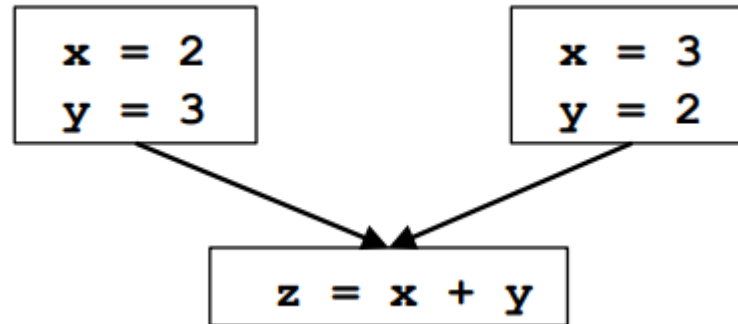
$f_s = \text{identity function}$

- Assignment $s: x = e$

$f_s(m) = m'$

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