Lecture 22

CIS 341: COMPILERS

Announcements

- HW5: Oat v. 2
  - Due *tomorrow* at midnight!
- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Available Thursday or Friday
  - Due: Wednesday, April 27th
- Final Exam:
  - According to registrar: Monday, May 2nd noon - 2:00pm

CODE ANALYSIS

Liveness information

- Consider this program:

```
int f(int x) {
  int a = x + 2;
  int b = a * a;
  int c = b + x;
  return c;
}
```

x is live

a and x are live

b and x are live

c is live

- The scopes of a,b,c,x all overlap – they're all in scope at the end of the block.
- But, a, b, c are never live at the same time.
  - So they can share the same stack slot / register

Live Variable Analysis

- A variable v is ***live*** at a program point if v is defined before the program point and used after it.
- Liveness is defined in terms of where variables are ***defined*** and where variables are ***used***

- Liveness analysis: Compute the live variables between each statement.
  - May be *conservative* (i.e. it may claim a variable is live when it isn't) so because that's a safe approximation
  - To be useful, it should be more *precise* than simple scoping rules.

- Liveness analysis is one example of *dataflow analysis*
  - Other examples: Available Expressions, Reaching Definitions, Constant-Propagation Analysis, …

Control-flow Graphs Revisited

- For the purposes of dataflow analysis, we use the *control-flow graph* (CFG) intermediate form.
- Recall that a basic block is a sequence of instructions such that:
  - There is a distinguished, labeled entry point (no jumps into the middle of a basic block)
  - There is a (possibly empty) sequence of non-control-flow instructions
  - The block ends with a single control-flow instruction (jump, conditional branch, return, etc.)

- A *control flow graph*
  - Nodes are blocks
  - There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
  - There are no "dangling" edges – there is a block for every jump target.

Note: the following slides are intentionally a bit ambiguous about the exact nature of the code in the control flow graphs:

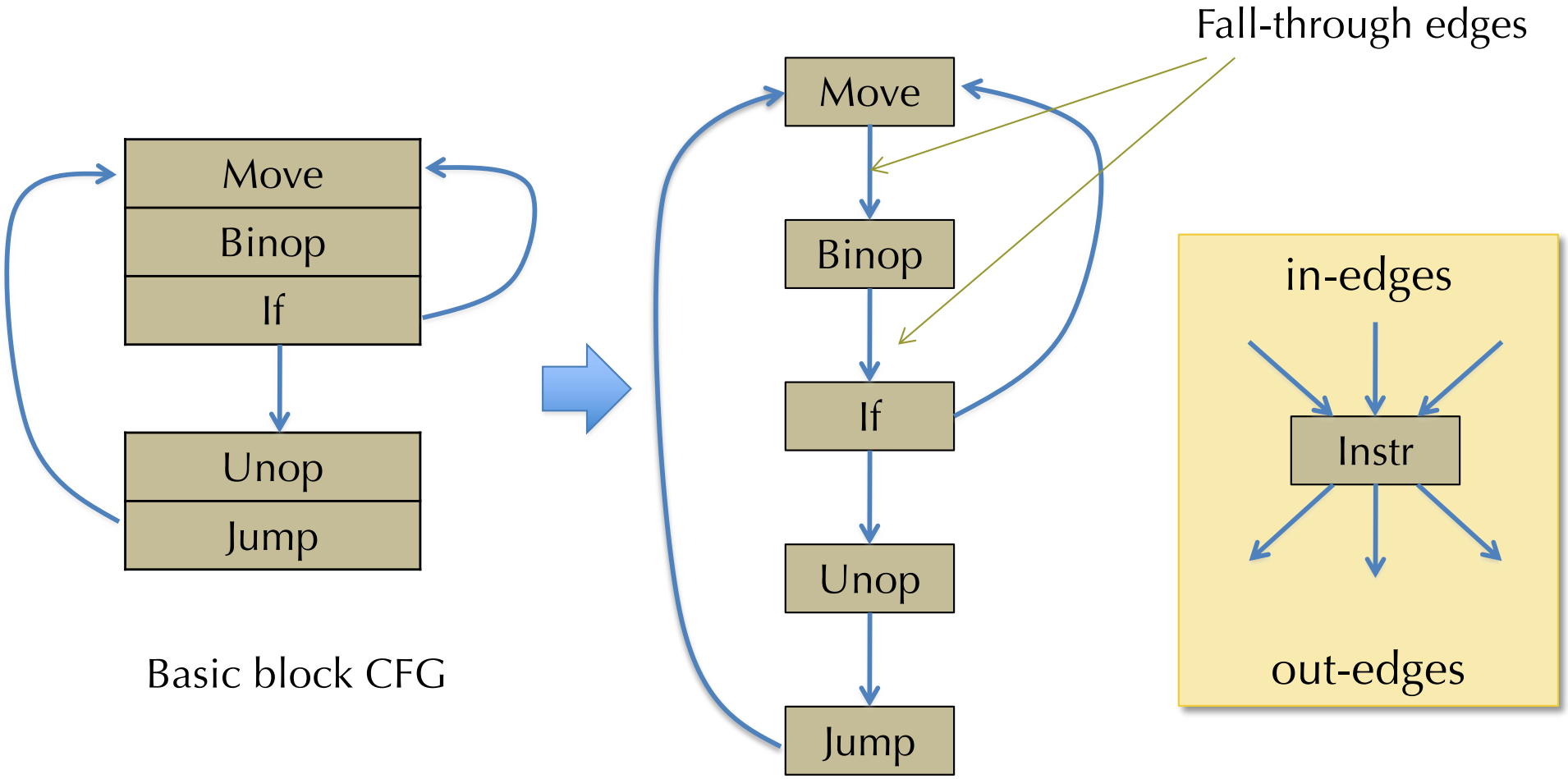
an "imperative" C-like source level
at the x86 assembly level
the LLVM IR level

Each setting applies the same general idea, but the exact details will differ.

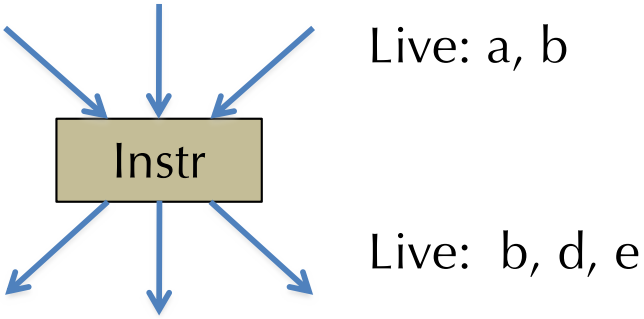
- e.g., LLVM IR doesn't have "imperative" update of %uid temporaries.
  (The SSA structure of the LLVM IR (by design!) makes some of these analyses simpler.)

Dataflow over CFGs

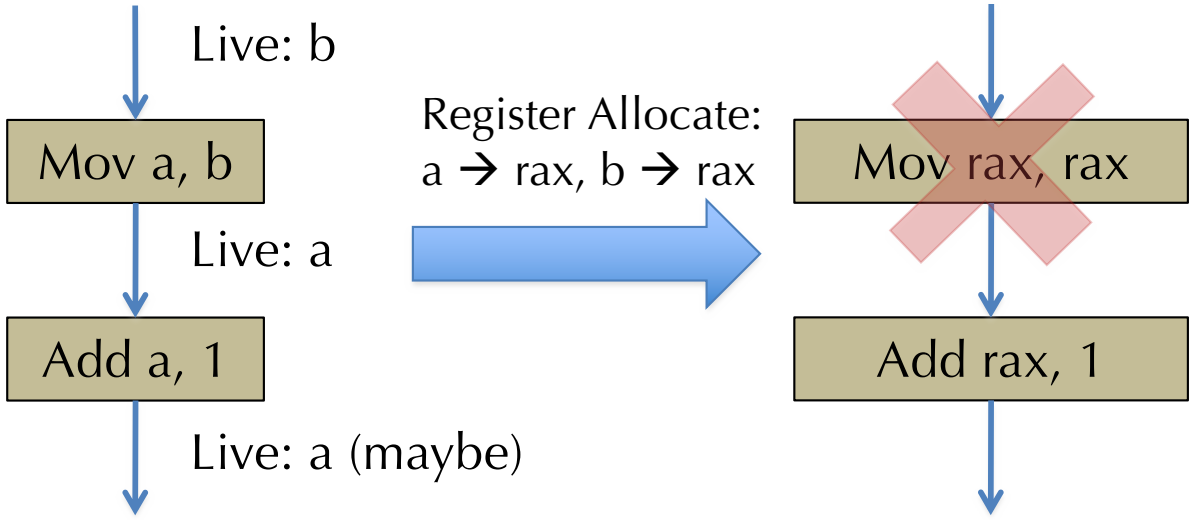
- For precision, it is helpful to think of the "fall through" between sequential instructions as an edge of the control-flow graph too.
  - Different implementation tradeoffs in practice…



Liveness is Associated with *Edges*



- This is useful so that the same register can be used for different temporaries in the same statement.
- Example: `a = b + 1`
- Compiles to:



Uses and Definitions

- Every instruction/statement *uses* some set of variables
  - i.e. reads from them
- Every instruction/statement *defines* some set of variables
  - i.e. writes to them

- For a node/statement s define:
  - use[s] : set of variables used by s
  - def[s] : set of variables defined by s

- Examples:
  - `a = b + c` use[s] = {b,c} def[s] = {a}
  - `a = a + 1` use[s] = {a} def[s] = {a}

Liveness, Formally

- A variable v is *live* on edge e if:
  There is
  - a node n in the CFG such that use[n] contains v, *and*
  - a directed path from e to n such that for every statement s′ on the path, def[s′] does not contain v

- The first clause says that v will be used on some path starting from edge e.
- The second clause says that v won't be redefined on that path before the use.

- Questions:
  - How to compute this efficiently?
  - How to use this information (e.g. for register allocation)?
  - How does the choice of IR affect this?
    (e.g. LLVM IR uses SSA, so it doesn't allow redefinition ⇒ simplify liveness analysis)

Simple, inefficient algorithm

- "A variable v is live on an edge e if there is a node n in the CFG using it *and* a directed path from e to n pasing through no def of v."

- Backtracking Algorithm:
  - For each variable v…
  - Try all paths from each use of v, tracing backwards through the control-flow graph until either v is defined or a previously visited node has been reached.
  - Mark the variable v live across each edge traversed.

- Inefficient because it explores the same paths many times (for different uses and different variables)

Dataflow Analysis

- *Idea*: compute liveness information for all variables simultaneously.
  - Keep track of sets of information about each node

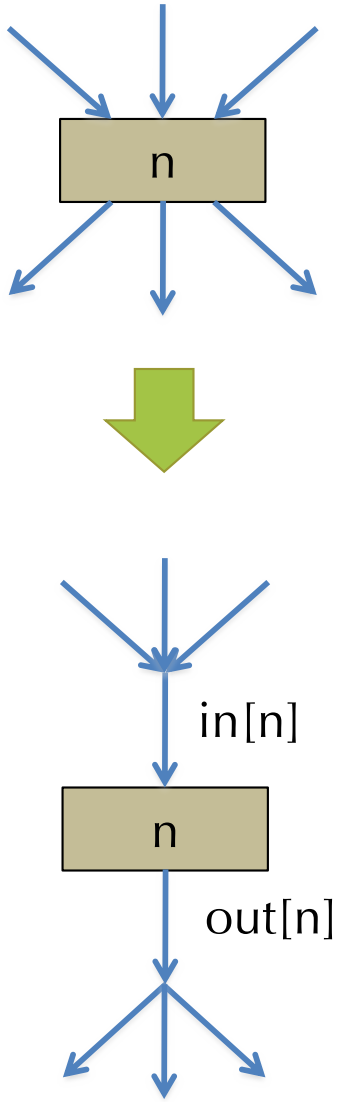
- Approach: define *equations* that must be satisfied by any liveness determination.
  - Equations based on "obvious" constraints.

- Solve the equations by iteratively converging on a solution.
  - Start with a "rough" approximation to the answer
  - Refine the answer at each iteration
  - Keep going until no more refinement is possible: a *fixpoint* has been reached

- This is an instance of a general framework for computing program properties: dataflow analysis

Dataflow Value Sets for Liveness

- Nodes are program statements, so:
- use[n] : set of variables used by n
- def[n] : set of variables defined by n
- in[n] : set of variables live on entry to n
- out[n] : set of variables live on exit from n



- Associate in[n] and out[n] with the "collected" information about incoming/outgoing edges

- For Liveness: what constraints are there among these sets?
- Clearly:

$$in[n] \supseteq use[n]$$

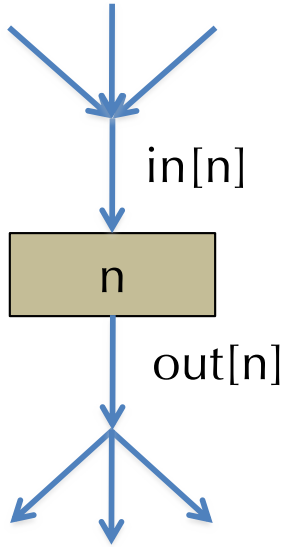
- What other constraints?

Other Dataflow Constraints

- We have: in[n] ⊇ use[n]
  - "A variable must be live on entry to n if it is used by n"

- Also: in[n] ⊇ out[n] - def[n]
  - "If a variable is live on exit from n, and n doesn't define it, it is live on entry to n"
  - Note: here '-' means "set difference"

- And: out[n] ⊇ in[n'] if n' ∈ succ[n]
  - "If a variable is live on entry to a successor node of n, it must be live on exit from n."



Iterative Dataflow Analysis

- Find a solution to those constraints by starting from a rough guess.
  - Start with: $in[n] = \emptyset$ and $out[n] = \emptyset$
- The guesses don't satisfy the constraints:
  - $in[n] \supseteq use[n]$
  - $in[n] \supseteq out[n] - def[n]$
  - $out[n] \supseteq in[n']$ if $n' \in succ[n]$

- Idea: iteratively re-compute in[n] and out[n] where forced to by the constraints.
  - Each iteration will add variables to the sets in[n] and out[n] (i.e. the live variable sets will increase monotonically)
- We stop when in[n] and out[n] satisfy these equations: (which are derived from the constraints above)
  - $in[n] = use[n] \cup (out[n] - def[n])$
  - $out[n] = \bigcup_{n' \in succ[n]} in[n']$

Complete Liveness Analysis Algorithm

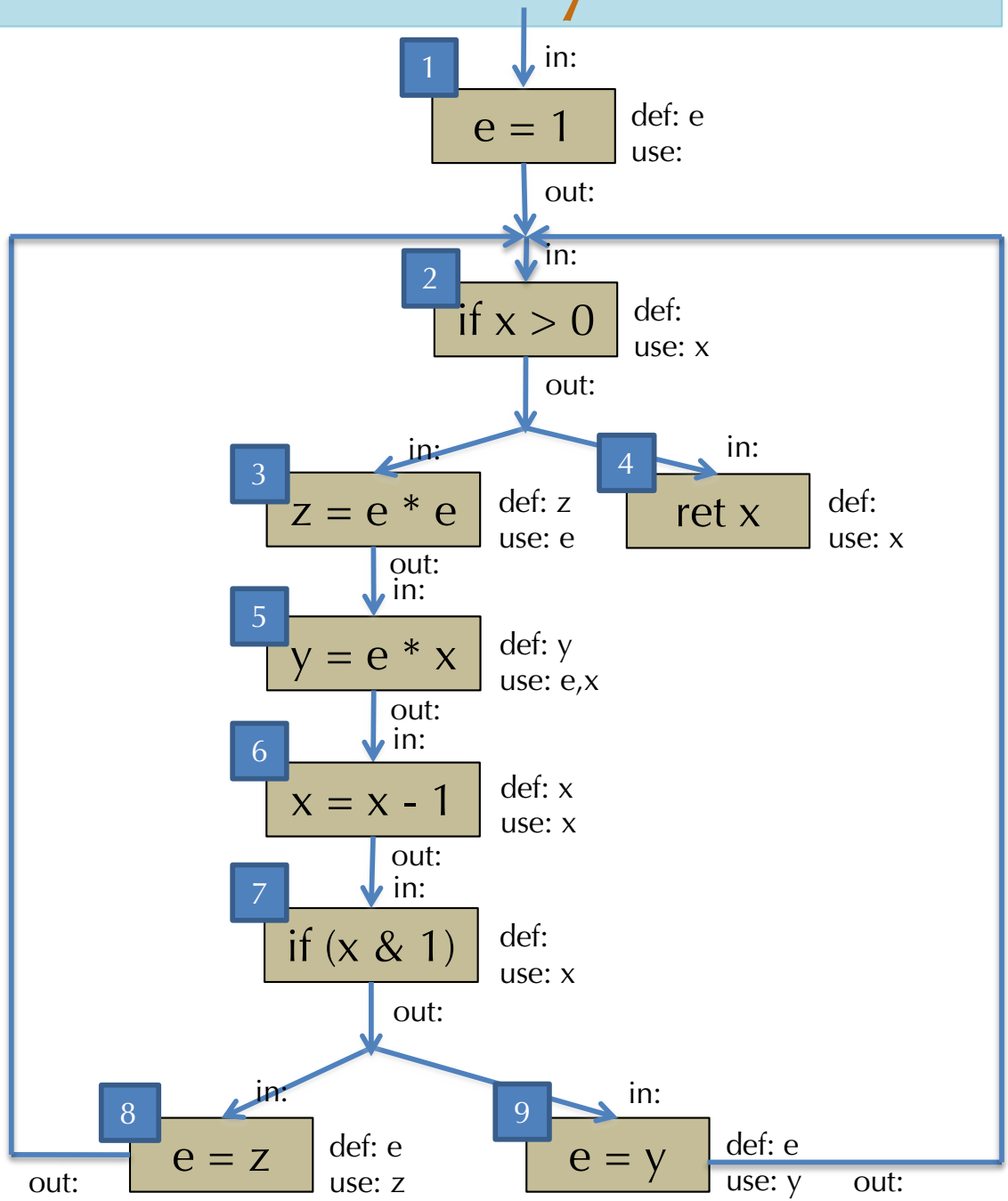
for all n, in[n] := Ø, out[n] := Ø
repeat until no change in 'in' and 'out'
    for all n
      out[n] := $\bigcup_{n' \in succ[n]}$ in[n']
      in[n] := use[n] ∪ (out[n] - def[n])
    end
end

- Finds a *fixpoint* of the in and out equations.
  - The algorithm is guaranteed to terminate… Why?
- Why do we start with Ø?

Example Liveness Analysis

- Example flow graph:

```
e = 1;
while(x>0) {
  z = e * e;
  y = e * x;
  x = x - 1;
  if (x & 1) {
    e = z;
  } else {
    e = y;
  }
}
return x;
```



Example Liveness Analysis

Each iteration update:

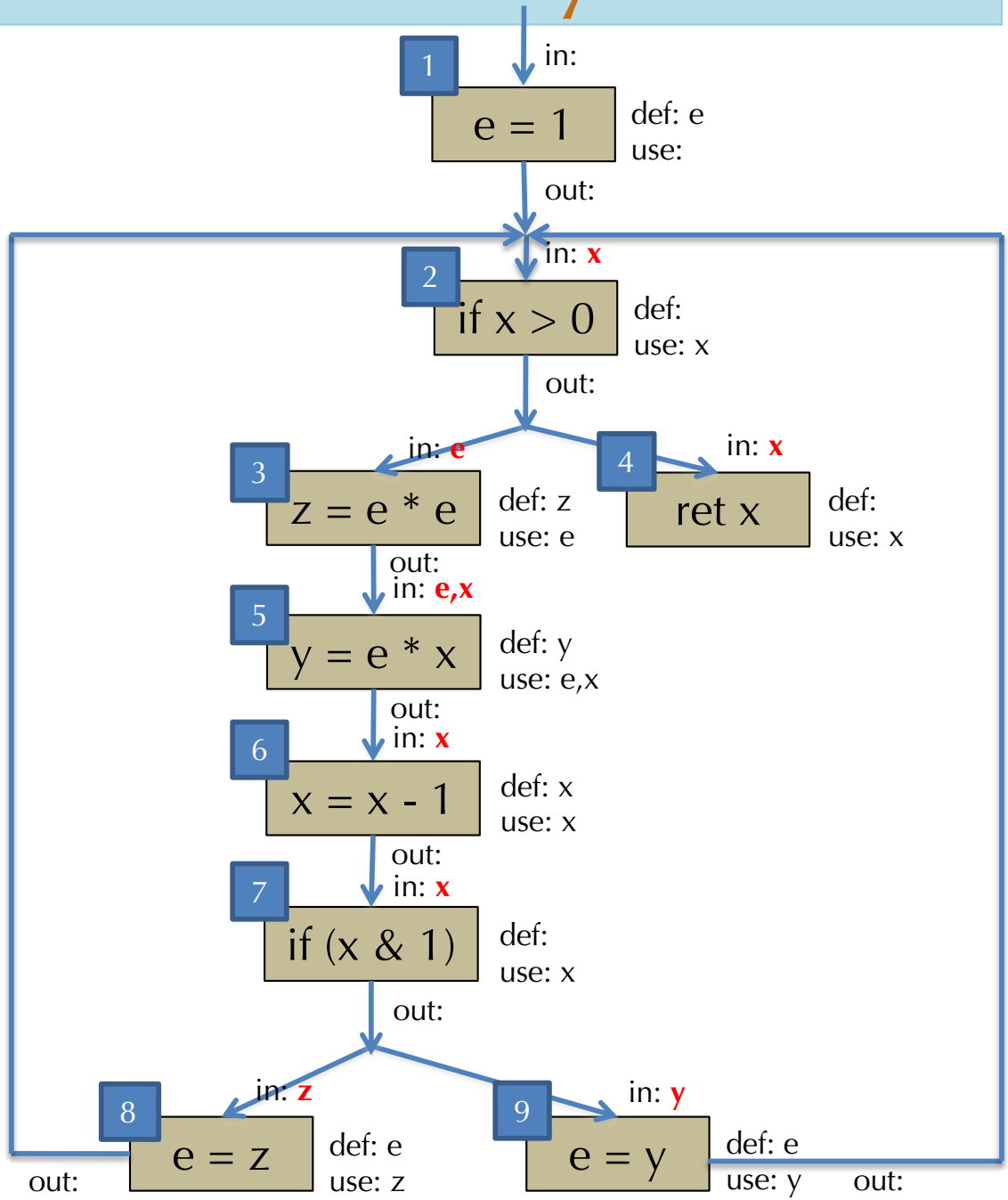
$out[n] := \bigcup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 1:

in[2] = x
in[3] = e
in[4] = x
in[5] = e,x
in[6] = x
in[7] = x
in[8] = z
in[9] = y

(showing only updates that make a change)



Example Liveness Analysis

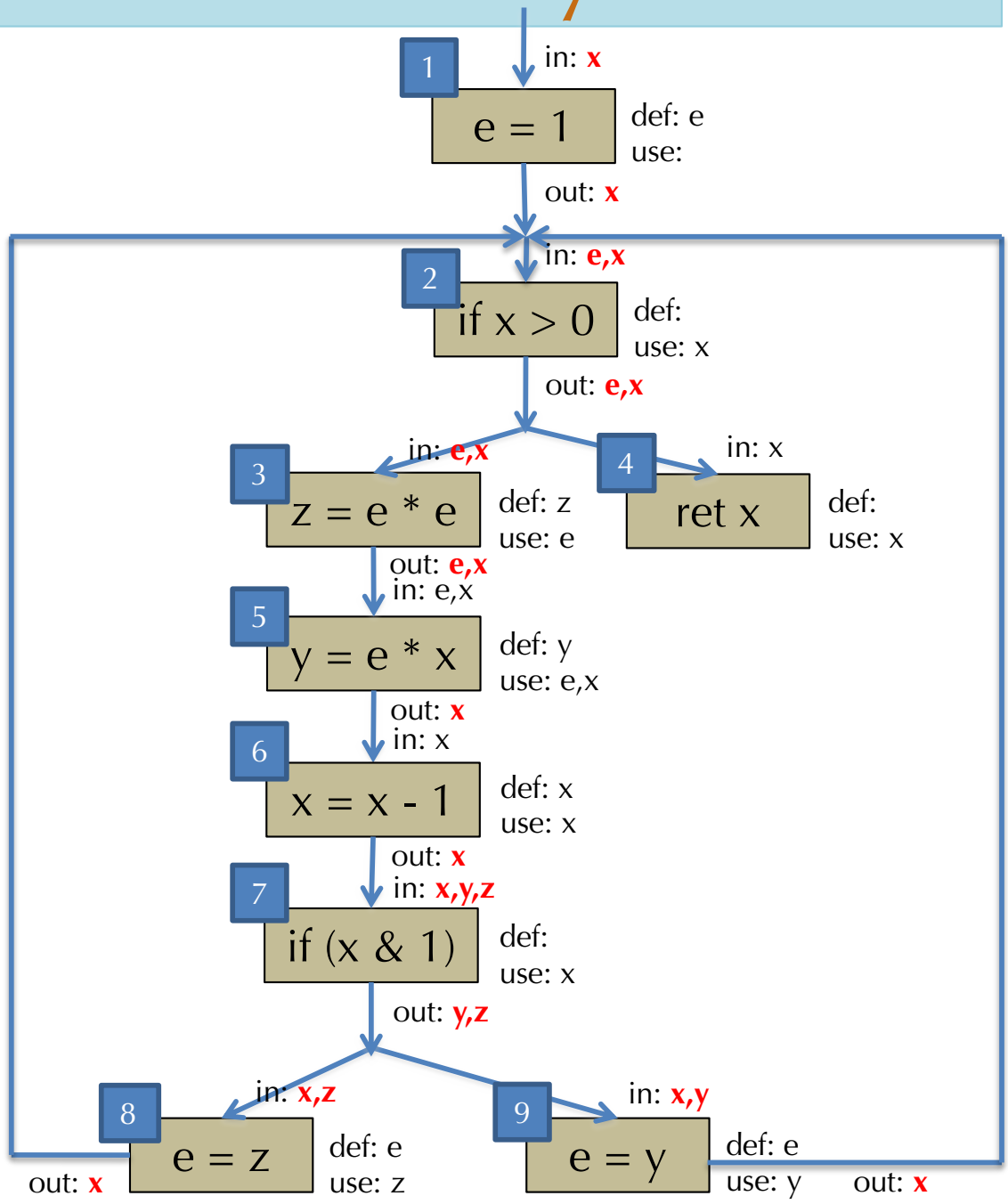
Each iteration update:

$out[n] := \cup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 2:

out[1]= x
in[1] = x
out[2] = e,x
in[2] = e,x
out[3] = e,x
in[3] = e,x
out[5] = x
out[6] = x
out[7] = z,y
in[7] = x,z,y
out[8] = x
in[8] = x,z
out[9] = x
in[9] = x,y



Example Liveness Analysis

Each iteration update:

$out[n] := \cup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 3:

out[1]= e,x

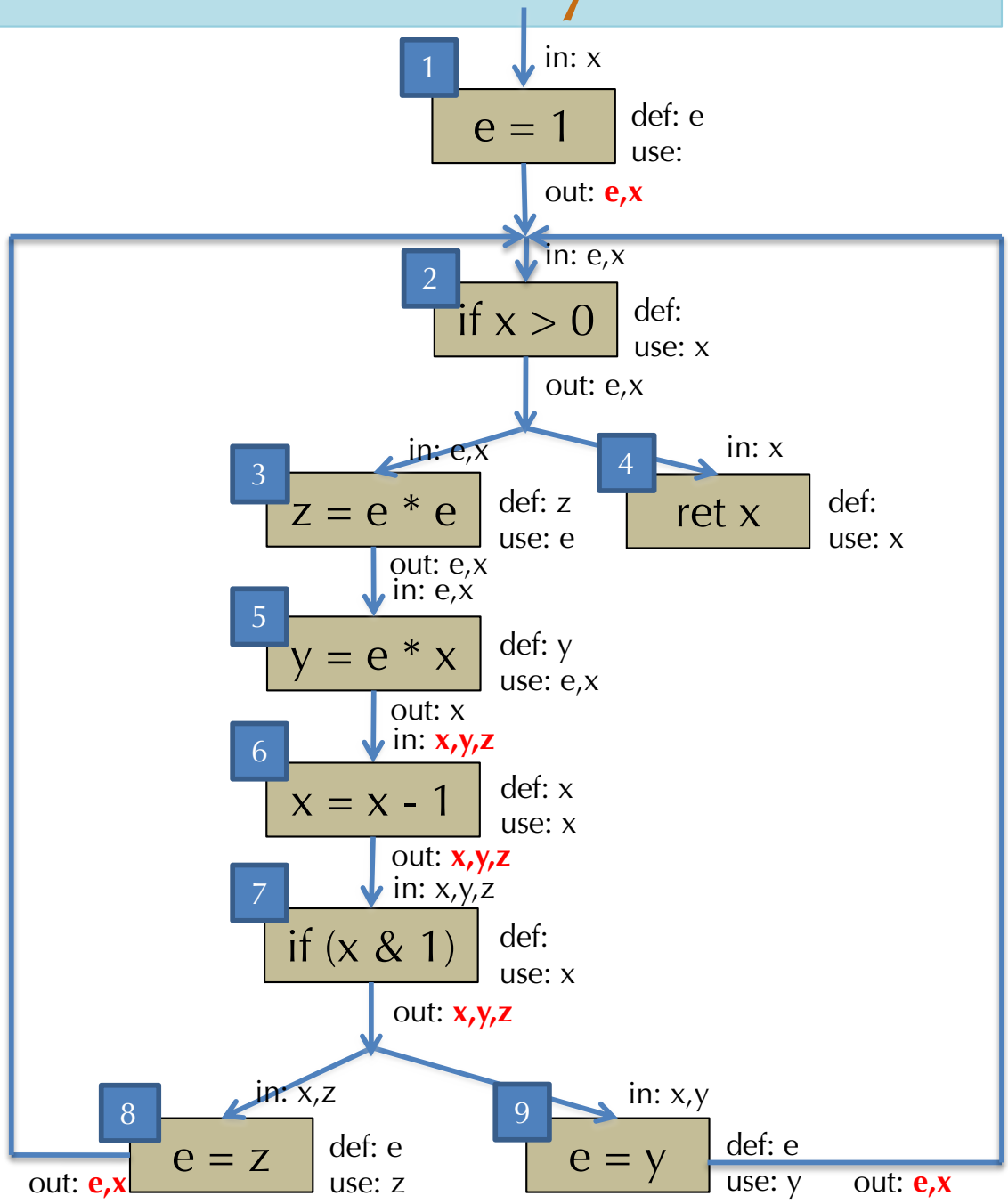
out[6]= x,y,z

in[6]= x,y,z

out[7]= x,y,z

out[8]= e,x

out[9]= e,x



Example Liveness Analysis

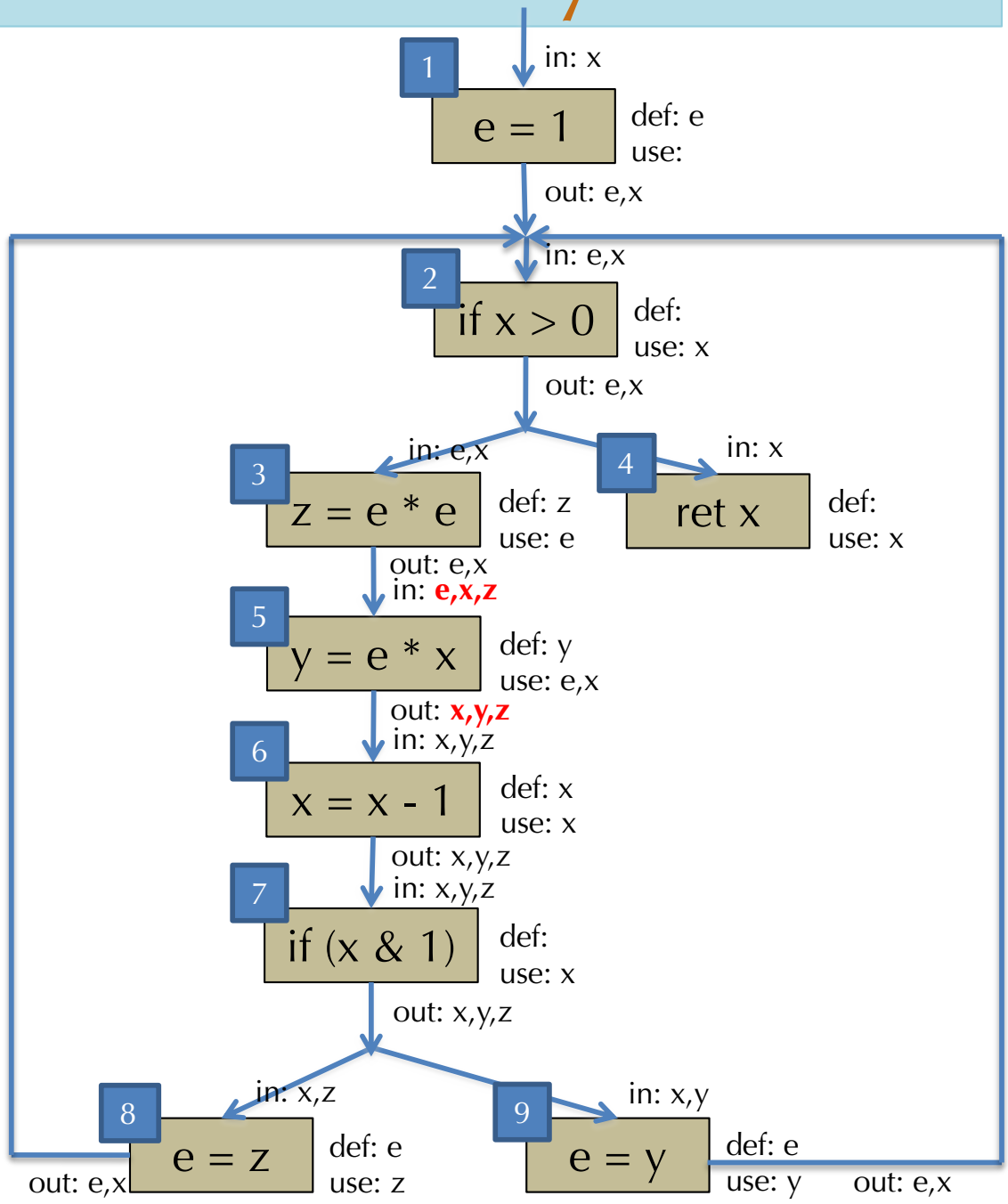
Each iteration update:

$out[n] := \cup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 4:

out[5]= x,y,z
in[5]= e,x,z



Example Liveness Analysis

Each iteration update:

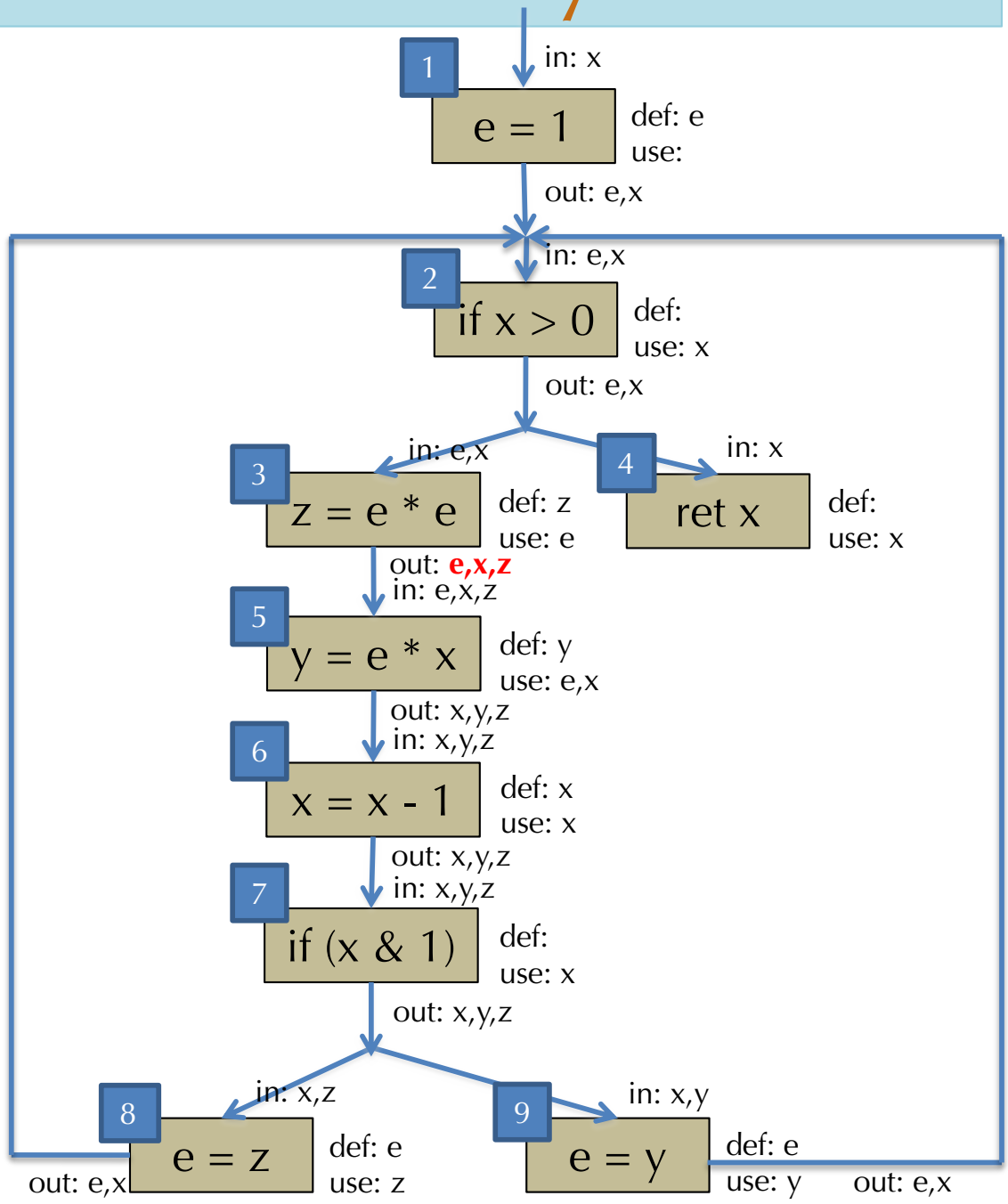
$out[n] := \cup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 5:

out[3]= e,x,z

Done!



Improving the Algorithm

- Can we do better?
- Observe: the only way information propagates from one node to another is using: $\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$
  - This is the only rule that involves more than one node
- If a node's successors haven't changed, then the node itself won't change.
- Idea for an improved version of the algorithm:
  - Keep track of which node's successors have changed

A Worklist Algorithm

- Use a FIFO queue of nodes that might need to be updated.

```
for all n, in[n] := Ø, out[n] := Ø
w = new queue with all nodes
repeat until w is empty
    let n = w.pop()                          // pull a node off the queue
     old_in = in[n]                          // remember old in[n]
     out[n] := ⋃_{n'∈succ[n]} in[n']
     in[n] := use[n] ∪ (out[n] - def[n])
     if (old_in != in[n]),                   // if in[n] has changed
        for all m in pred[n], w.push(m)      // add to worklist
end
```

OTHER DATAFLOW ANALYSES

Generalizing Dataflow Analyses

- The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well.
  - Reaching definitions analysis
  - Available expressions analysis
  - Alias Analysis
  - Constant Propagation
  - These analyses follow the same 3-step approach as for liveness.

- To see these as an instance of the same kind of algorithm, the next few examples to work over a canonical intermediate instruction representation called *quadruples*
  - Allows easy definition of def[n] and use[n]
  - A slightly "looser" variant of LLVM's IR that doesn't require the "static single assignment" – i.e. it has *mutable* local variables
  - We will use LLVM-IR-like syntax

Def / Use for SSA

Instructions n:	def[n]	use[n]	description
a = op b c	{a}	{b,c}	arithmetic
a = load b	{a}	{b}	load
store a, b	Ø	{b}	store
a = alloca t	{a}	Ø	alloca
a = bitcast b to u	{a}	{b}	bitcast
a = gep b [c,d, …]	{a}	{b,c,d,…}	getelementptr
a = f(b_1,…,b_n)	{a}	{b_1,…,b_n}	call w/return
f(b_1,…,b_n)	Ø	{b_1,…,b_n}	void call (no return)

- Terminators

br L	Ø	Ø	jump
br a L1 L2	Ø	{a}	conditional branch
return a	Ø	{a}	return

REACHING DEFINITIONS

Reaching Definition Analysis

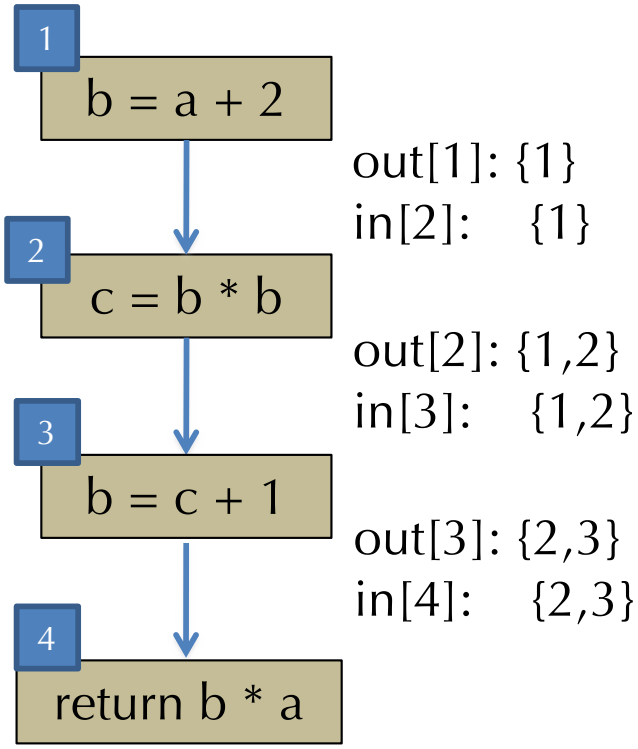
- Question: what uses in a program does a given variable definition reach?

- This analysis is used for constant propagation & copy prop.
  - If only one definition reaches a particular use, can replace use by the definition (for constant propagation).
  - Copy propagation additionally requires that the copied value still has its same value – computed using an *available expressions* analysis (next)

- Input: Quadruple CFG
- Output: in[n] (resp. out[n]) is the set of nodes defining some variable such that the definition may reach the beginning (resp. end) of node n

Example of Reaching Definitions

- Results of computing reaching definitions on this simple CFG:



Reaching Definitions Step 1

- Define the sets of interest for the analysis
- Let defs[a] be the set of *nodes* that define the variable a
- Define gen[n] and kill[n] as follows:
- Quadruple forms n:

Quadruple forms n:	gen[n]	kill[n]
a = b op c	{n}	defs[a] - {n}
a = load b	{n}	defs[a] - {n}
store b, a	Ø	Ø
a = f(b_1,…,b_n)	{n}	defs[a] - {n}
f(b_1,…,b_n)	Ø	Ø
br L	Ø	Ø
br a L1 L2	Ø	Ø
return a	Ø	Ø

Reaching Definitions Step 2

- Define the constraints that a reaching definitions solution must satisfy.
- out[n] ⊇ gen[n]
  "The definitions that reach the end of a node at least include the definitions generated by the node"

- in[n] ⊇ out[n'] if n' is in pred[n]
  "The definitions that reach the beginning of a node include those that reach the exit of *any* predecessor"

- out[n] ∪ kill[n] ⊇ in[n]
  "The definitions that come in to a node either reach the end of the node or are killed by it."
  - Equivalently: out[n] ⊇ in[n] - kill[n]

Reaching Definitions Step 3

- Convert constraints to iterated update equations:
- $in[n] := \bigcup_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] - kill[n])$

- Algorithm: initialize $in[n]$ and $out[n]$ to $\emptyset$
  - Iterate the update equations until a fixed point is reached

- The algorithm terminates because in[n] and out[n] increase only *monotonically*
  - At most to a maximum set that includes all variables in the program
- The algorithm is precise because it finds the *smallest* sets that satisfy the constraints.

AVAILABLE EXPRESSIONS

Available Expressions

- Idea: want to perform common subexpression elimination:
  - a = x + 1 … b = x + 1  a = x + 1 … b = a
- This transformation is safe if x+1 means computes the same value at both places (i.e. x hasn't been assigned).
  - "x+1" is an *available expression*

- Dataflow values:
  - in[n] = set of nodes whose values are available on entry to n
  - out[n] = set of nodes whose values are available on exit of n

Available Expressions Step 1

- Define the sets of values
- Define gen[n] and kill[n] as follows:
- Quadruple forms n:

Quadruple forms n:	gen[n]	kill[n]
a = b op c	{n} - kill[n]	uses[a]
a = load b	{n} - kill[n]	uses[a]
store b, a	∅	uses[[x]] (for all x that may equal a)
br L	∅	∅
br a L1 L2	∅	∅
a = f(b_1,…,b_n)	∅	uses[a]∪ uses[[x]] (for all x)
f(b_1,…,b_n)	∅	uses[[x]] (for all x)
return a	∅	∅

Note the need for "may alias" information…

Note that functions are assumed to be impure…

Available Expressions Step 2

- Define the constraints that an available expressions solution must satisfy.
- $out[n] \supseteq gen[n]$
  "The expressions made available by n that reach the end of the node"

- $in[n] \subseteq out[n']$ if n' is in $pred[n]$
  "The expressions available at the beginning of a node include those that reach the exit of *every* predecessor"

- $out[n] \cup kill[n] \supseteq in[n]$
  "The expressions available on entry either reach the end of the node or are killed by it."
  - Equivalently: $out[n] \supseteq in[n] - kill[n]$

Note similarities and differences with constraints for "reaching definitions".

Available Expressions Step 3

- Convert constraints to iterated update equations:
- $in[n] := \bigcap_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] - kill[n])$

- Algorithm: initialize in[n] and out[n] to {set of all nodes}
  - Iterate the update equations until a fixed point is reached

- The algorithm terminates because in[n] and out[n] *decrease* only *monotonically*
  - At most to a minimum of the empty set
- The algorithm is precise because it finds the *largest* sets that satisfy the constraints.

GENERAL DATAFLOW ANALYSIS

Comparing Dataflow Analyses

- Look at the update equations in the inner loop of the analyses
- Liveness: (backward)
  - Let gen[n] = use[n] and kill[n] = def[n]
  - out[n] := = $\bigcup_{n' \in succ[n]}$in[n′]
  - in[n] := gen[n] ∪ (out[n] - kill[n])
- Reaching Definitions: (forward)
  - in[n] := $\bigcup_{n' \in pred[n]}$out[n′]
  - out[n] := gen[n] ∪ (in[n] - kill[n])
- Available Expressions: (forward)
  - in[n] := $\bigcap_{n' \in pred[n]}$out[n′]
  - out[n] := gen[n] ∪ (in[n] - kill[n])

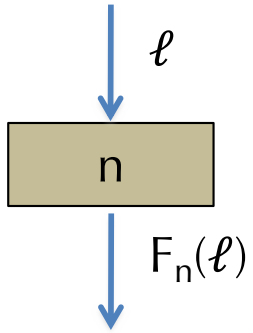
Common Features

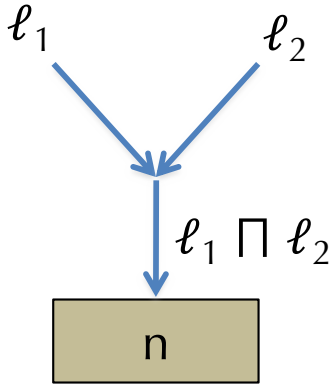
- All of these analyses have a *domain* over which they solve constraints.
  - Liveness, the domain is sets of variables
  - Reaching defns., Available exprs. the domain is sets of nodes
- Each analysis has a notion of gen[n] and kill[n]
  - Used to explain how information propagates across a node.
- Each analysis is propagates information either *forward* or *backward*
  - Forward: in[n] defined in terms of predecessor nodes' out[]
  - Backward: out[n] defined in terms of successor nodes' in[]
- Each analysis has a way of aggregating information
  - Liveness & reaching definitions take union (∪)
  - Available expressions uses intersection (∩)
  - Union expresses a property that holds for *some* path (existential)
  - Intersection expresses a property that holds for *all* paths (universal)

(Forward) Dataflow Analysis Framework

A forward dataflow analysis can be characterized by:

1. A domain of dataflow values $\mathcal{L}$
   - e.g. $\mathcal{L}$ = the powerset of all variables
   - Think of $\ell \in \mathcal{L}$ as a property, then "x ∈ ℓ" means "x has the property"
2. For each node n, a flow function $F_n : \mathcal{L} \rightarrow \mathcal{L}$
   - So far we've seen $F_n(\ell) = gen[n] \cup (\ell - kill[n])$
   - So: $out[n] = F_n(in[n])$
   - "If ℓ is a property that holds before the node n, then $F_n(\ell)$ holds after n"
3. A combining operator $\prod$
   - "If we know *either* ℓ_1 *or* ℓ_2 holds on entry to node n, we know at most $\ell_1 \prod \ell_2$"
   - $in[n] := \prod_{n' \in pred[n]} out[n']$





Generic Iterative (Forward) Analysis

for all n, in[n] := $\top$, out[n] := $\top$
repeat until no change
    for all n

$$in[n] := \bigsqcap_{n' \in pred[n]} out[n']$$

$$out[n] := F_n(in[n])$$

    end
end

- Here, $\top \in \mathcal{L}$ ("top") represents having the "maximum" amount of information.
  - Having "more" information enables more optimizations
  - "Maximum" amount could be inconsistent with the constraints.
  - Iteration refines the answer, eliminating inconsistencies

Structure of $\mathcal{L}$

- The domain has structure that reflects the "amount" of information contained in each dataflow value.
- Some dataflow values are more informative than others:
  - Write $\ell_1 \sqsubseteq \ell_2$ whenever ℓ_2 provides at least as much information as ℓ_1.
  - The dataflow value ℓ_2 is "better" for enabling optimizations.

- Example 1: for liveness analysis, *smaller* sets of variables are more informative.
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments.
  - So: $\ell_1 \sqsubseteq \ell_2$ if and only if $\ell_1 \supseteq \ell_2$

- Example 2: for available expressions analysis, larger sets of nodes are more informative.
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination.
  - So: $\ell_1 \sqsubseteq \ell_2$ if and only if $\ell_1 \subseteq \ell_2$

$\mathcal{L}$ as a Partial Order

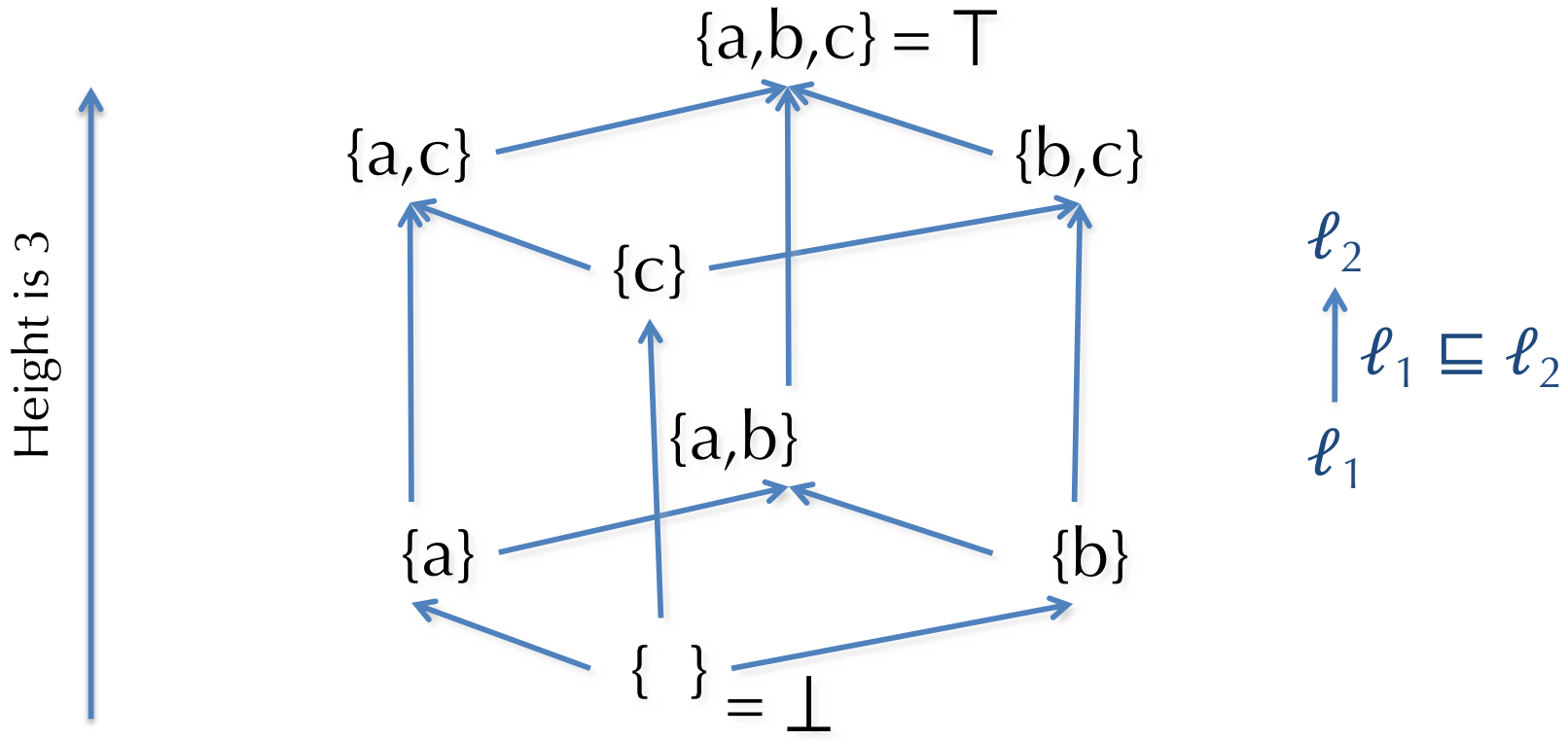
- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq \ell_1$

- Properties of a partial order:
  - *Reflexivity:* $\ell \sqsubseteq \ell$
  - *Transitivity:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_3$ implies $\ell_1 \sqsubseteq \ell_2$
  - *Anti-symmetry:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$ implies $\ell_1 = \ell_2$

- Examples:
  - Integers ordered by $\leq$
  - Types ordered by <:
  - Sets ordered by $\subseteq$ or $\supseteq$

Subsets of {a,b,c} ordered by ⊆

Partial order presented as a Hasse diagram.



order ⊑ is ⊆ meet ⊓ is ∩ join ⊔ is ∪

Meets and Joins

- The combining operator $\sqcap$ is called the "meet" operation.
- It constructs the *greatest lower bound*:
  - $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_1$ and $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_2$
    "the meet is a lower bound"
  - If $\ell \sqsubseteq \ell_1$ and $\ell \sqsubseteq \ell_2$ then $\ell \sqsubseteq \ell_1 \sqcap \ell_2$
    "there is no greater lower bound"

- Dually, the $\sqcup$ operator is called the "join" operation.
- It constructs the *least upper bound*:
  - $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$
    "the join is an upper bound"
  - If $\ell_1 \sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$ then $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$
    "there is no smaller upper bound"

- A partial order that has all meets and joins is called a *lattice*.
  - If it has just meets, it's called a *meet semi-lattice*.

Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node n):
- $out[n] := F_n(in[n])$

- Equivalently: $out[n] := F_n(\bigsqcap_{n' \in pred[n]} out[n'])$
  - By definition of in[n]
- We can write this as a simultaneous update of the vector of out[n] values:
  - let $x_n = out[n]$
  - Let $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ it's a vector of points in $\mathcal{L}$
  - $\mathbf{F}(\mathbf{X}) = (F_1(\bigsqcap_{j \in pred[1]} out[j]), F_2(\bigsqcap_{j \in pred[2]} out[j]), \ldots, F_n(\bigsqcap_{j \in pred[n]} out[j]))$

- Any solution to the constraints is a *fixpoint* **X** of **F**
  - i.e. $\mathbf{F}(\mathbf{X}) = \mathbf{X}$

Iteration Computes Fixpoints

- Let $\mathbf{X}_0 = (\top, \top, \ldots, \top)$
- Each loop through the algorithm apply F to the old vector:
  $\mathbf{X}_1 = \mathbf{F}(\mathbf{X}_0)$
  $\mathbf{X}_2 = \mathbf{F}(\mathbf{X}_1)$
  …
- $\mathbf{F}^{k+1}(\mathbf{X}) = \mathbf{F}(\mathbf{F}^{k}(\mathbf{X}))$
- A fixpoint is reached when $\mathbf{F}^{k}(\mathbf{X}) = \mathbf{F}^{k+1}(\mathbf{X})$
  - That's when the algorithm stops.

- Wanted: a maximal fixpoint
  - Because that one is more informative/useful for performing optimizations

Monotonicity & Termination

- Each flow function F_n maps lattice elements to lattice elements; to be sensible is should be *monotonic*:
- $F : \mathcal{L} \rightarrow \mathcal{L}$ is *monotonic* iff:
  $\ell_1 \sqsubseteq \ell_2$ implies that $F(\ell_1) \sqsubseteq F(\ell_2)$
  - Intuitively: "If you have more information entering a node, then you have more information leaving the node."

- Monotonicity lifts point-wise to the function: $\mathbf{F} : \mathcal{L}^n \rightarrow \mathcal{L}^n$
  - vector $(x_1, x_2, \ldots, x_n) \sqsubseteq (y_1, y_2, \ldots, y_n)$ iff $x_i \sqsubseteq y_i$ for each i

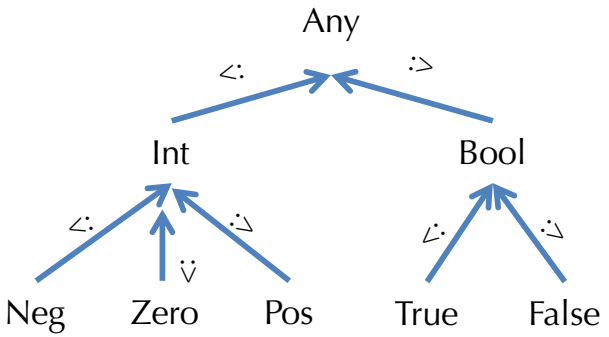
- Note that **F** is consistent: $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\ldots \sqsubseteq \mathbf{F}(\mathbf{F}(\mathbf{X}_0)) \sqsubseteq \mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$

- Therefore, # steps needed to reach a fixpoint is at most the height H of $\mathcal{L}$ times the number of nodes: O(Hn)

Building Lattices?

- Information about individual nodes or variables can be lifted *pointwise:*
  - If $\mathcal{L}$ is a lattice, then so is $\{ f : X \rightarrow \mathcal{L} \}$ where $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$.

- Like *types*, the dataflow lattices are *static approximations* to the dynamic behavior:
  - Could pick a lattice based on subtyping:



  - Or other information:

- Points in the lattice are sometimes called dataflow "facts"

"Classic" Constant Propagation

- Constant propagation can be formulated as a dataflow analysis.

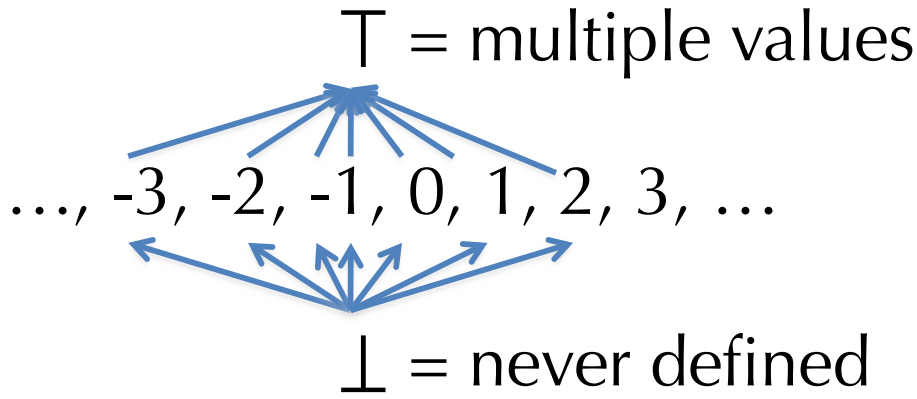
- Idea: propagate and fold integer constants in one pass:

```
x = 1;              x = 1;
y = 5 + x;    ⇒     y = 6;
z = y * y;          z = 36;
```

- Information about a single variable:
  - Variable is never defined.
  - Variable has a single, constant value.
  - Variable is assigned multiple values.

Domains for Constant Propagation

- We can make a constant propagation lattice $\mathcal{L}$ for *one variable* like this:



- To accommodate multiple variables, we take the product lattice, with one element per variable.
  - Assuming there are three variables, x, y, and z, the elements of the product lattice are of the form (ℓ_x, ℓ_y, ℓ_z).
  - Alternatively, think of the product domain as a context that maps variable names to their "*abstract interpretations*"
- What are "meet" and "join" in this product lattice?
- What is the height of the product lattice?

Flow Functions

- Consider the node `x = y op z`
- $F(\ell_x, \ell_y, \ell_z) = ?$

- $F(\ell_x, \top, \ell_z) = (\top, \top, \ell_z)$
- $F(\ell_x, \ell_y, \top) = (\top, \ell_y, \top)$

"If either input might have multiple values the result of the operation might too."

- $F(\ell_x, \bot, \ell_z) = (\bot, \bot, \ell_z)$
- $F(\ell_x, \ell_y, \bot) = (\bot, \ell_y, \bot)$

"If either input is undefined the result of the operation is too."

- $F(\ell_x, i, j) = (i \text{ op } j, i, j)$

"If the inputs are known constants, calculate the output statically."

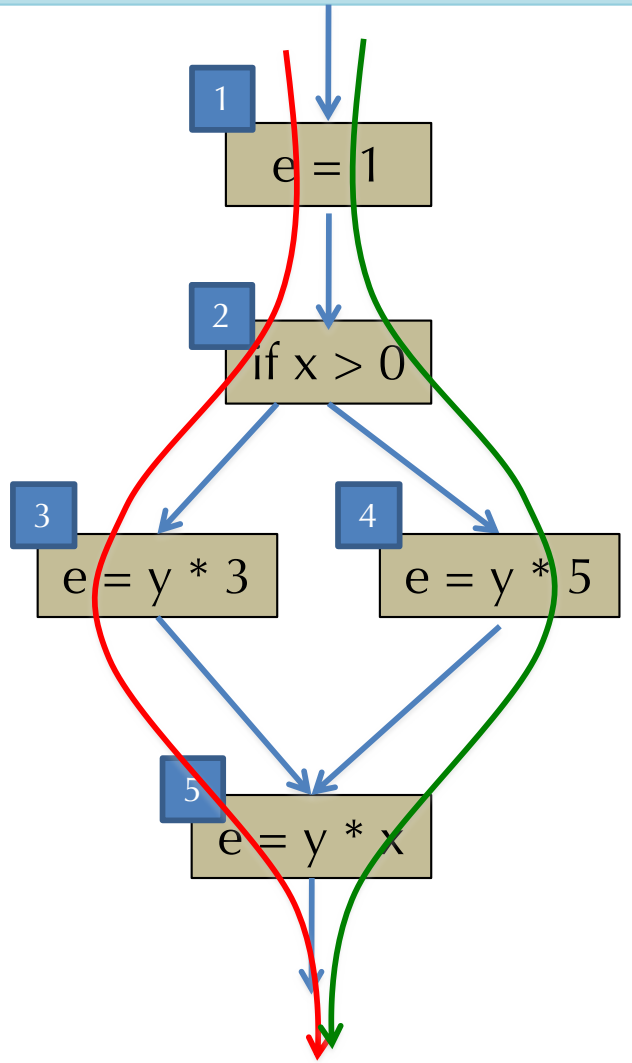
- Flow functions for the other nodes are easy…
- Monotonic?
- Distributes over meets?

QUALITY OF DATAFLOW ANALYSIS SOLUTIONS

Best Possible Solution

- Suppose we have a control-flow graph.
- If there is a path p_1 starting from the root node (entry point of the function) traversing the nodes
  $n_0, n_1, n_2, \ldots n_k$
- The best possible information along the path p_1 is:
  $\ell_{p1} = F_{nk}(\ldots F_{n2}(F_{n1}(F_{n0}(T)))\ldots)$
- Best solution at the output is some $\ell \sqsubseteq \ell_p$ for *all* paths p.

- Meet-over-paths (MOP) solution:
  $\prod_{p \in paths\_to[n]} \ell_p$



Best answer here is:

$F_5(F_3(F_2(F_1(T)))) \sqcap F_5(F_4(F_2(F_1(T))))$

What about quality of iterative solution?

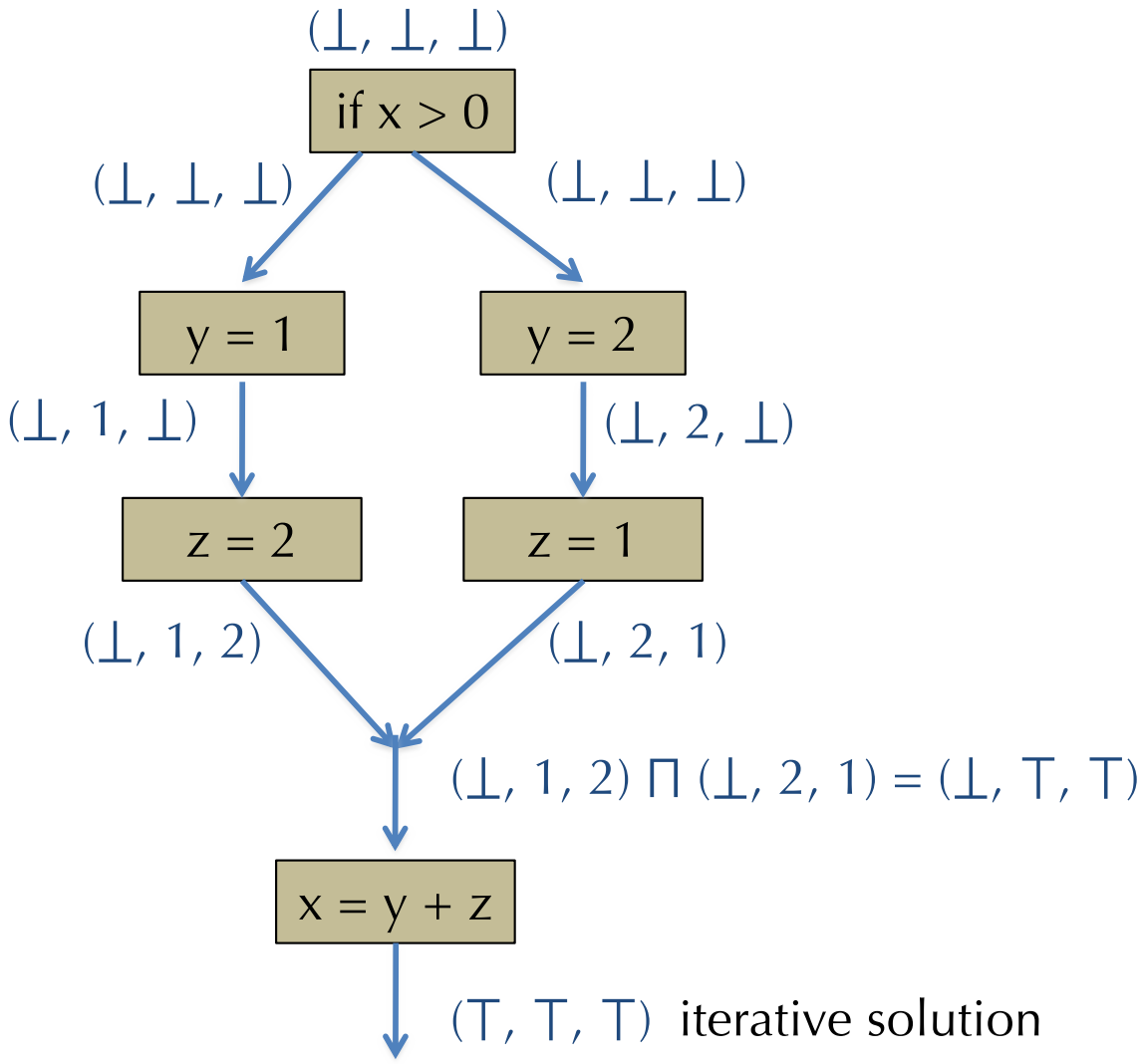
- Does the iterative solution: $out[n] = F_n(\bigsqcap_{n' \in pred[n]} out[n'])$ compute the MOP solution?
- MOP Solution: $\bigsqcap_{p \in paths\_to[n]} \ell_p$
- Answer: Yes, *if* the flow functions *distribute* over $\bigsqcap$
  - Distributive means: $\bigsqcap_i F_n(\ell_i) = F_n(\bigsqcap_i \ell_i)$
  - Proof is a bit tricky & beyond the scope of this class. (Difficulty: loops in the control flow graph might mean there are *infinitely* many paths…)
- Not all analyses give MOP solution
  - They are more conservative.

Reaching Definitions is MOP

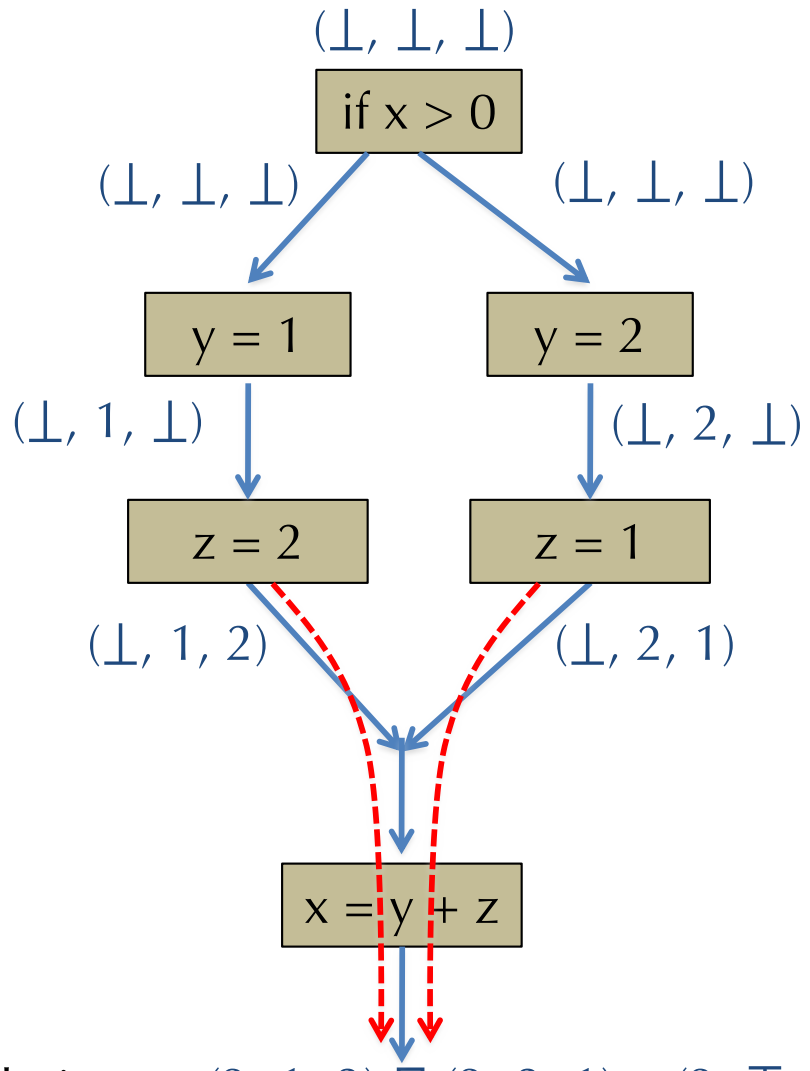
- $F_n[x] = gen[n] \cup (x - kill[n])$
- Does F_n distribute over meet $\sqcap = \cup$?
- $F_n[x \sqcap y]$
  $= gen[n] \cup ((x \cup y) - kill[n])$
  $= gen[n] \cup ((x - kill[n]) \cup (y - kill[n]))$
  $= (gen[n] \cup (x - kill[n])) \cup (gen[n] \cup (y - kill[n]))$
  $= F_n[x] \cup F_n[y]$
  $= F_n[x] \sqcap F_n[y]$

- Therefore: Reaching Definitions with iterative analysis always terminates with the MOP (i.e. best) solution.

Constprop Iterative Solution



MOP Solution ≠ Iterative Solution

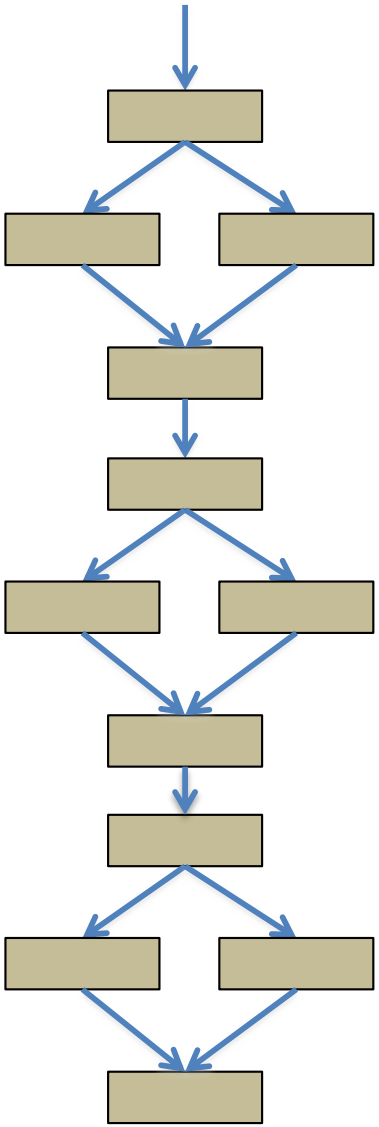


MOP solution $(3, 1, 2) \sqcap (3, 2, 1) = (3, \top, \top)$

Why not compute MOP Solution?

- If MOP is better than the iterative analysis, why not compute it instead?
  - ANS: exponentially many paths (even in graph without loops)

- O(n) nodes
- O(n) edges
- $O(2^n)$ paths*
  - At each branch there is a choice of 2 directions



* Incidentally, a similar idea can be used to force ML / Haskell type inference to need to construct a type that is exponentially big in the size of the program!

Dataflow Analysis: Summary

- Many dataflow analyses fit into a common framework.
- Key idea: *Iterative solution* of a system of equations over a *lattice* of constraints.
  - Iteration terminates if flow functions are monotonic.
  - Solution is equivalent to meet-over-paths answer if the flow functions distribute over meet ($\sqcap$).

- Dataflow analyses as presented work for an "imperative" intermediate representation.
  - The values of temporary variables are updated ("mutated") during evaluation.
  - Such mutation complicates calculations
  - SSA = "Single Static Assignment" eliminates this problem, by introducing more temporaries – each one assigned to only once.
  - Next up: Converting to SSA, finding loops and dominators in CFGs

See HW6: Dataflow Analysis

IMPLEMENTATION