YSC4230: Programming Language Design and Implementation

Week 12: A General Framework for Dataflow Analysis

Ilya Sergey

ilya.sergey@yale-nus.edu.sg

The Zoo of Optimizations

Dead Code Elimination

- If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x  = y * y          // x is dead!
…                   // x never used  ➔       …
x = z * z                                      x = z * z
```

- A variable is *dead* if it is never used after it is defined.
  - Computing such *definition* and *use* information is an important component of compiler

- Dead variables can be created by other optimizations…

Unreachable/Dead Code

- Basic blocks not reachable by any trace leading from the starting basic block are *unreachable* and can be deleted.
  - Performed at the IR or assembly level

- Dead code: similar to unreachable blocks.
  - A value might be computed but never subsequently used.

- Code for computing the value can be dropped

- But only if it's *pure*, i.e. it has *no externally visible side effects*
  - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
  - Note: Pure functional languages (e.g. Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

Inlining

- Replace a call with the body of the function itself with arguments rewritten to be local variables:
- Example in Oat code:

```
int g(int x) { return x + pow(x); }
int pow(int a) { int b = 1; int n = 0;
                 while (n < a) {b = 2 * b};
                 return b; }
➔
int g(int x) {
  int a = x; int b = 1; int n = 0;
  while (n < a) {b = 2 * b}; tmp = b;
  return x + tmp;
}
```

- May need to rename variable names to avoid *name capture*
  - Example of what can go wrong?
- Best done at the AST or relatively high-level IR.
- When is it profitable?
  - Eliminates the stack manipulation, jump, etc.
  - Can increase code size.
  - Enables further optimizations

```
int g(int x) ( 1 + f(x) )
Int f(int a) (a + x)
➔
const int x = 3;
int g(int x) ( 1 + (int a = x; a + x) )
```

Code Specialization

- Idea: create specialized versions of a function that is called from different places with different arguments.
- Example: specialize function f in:

```
class A implements I { int m() {…} }
class B implements I { int m() {…} }
int f(I x) { x.m(); }                 // don't know which m
A a = new A(); f(a);                  // know it's A.m
B b = new B(); f(b);                  // know it's B.m
```

- f_A would have code specialized to dispatch to A.m
- f_B would have code specialized to dispatch to B.m
- You can also *inline* methods when the run-time type is known statically
  - Often just one class implements a method.

Common Subexpression Elimination (CSE)

- In some sense it's the opposite of inlining: fold redundant computations together
- Example:

  a[i] = a[i] + 1 compiles to:

  [a + i*4] = [a + i*4] + 1

Common subexpression elimination removes the redundant add and multiply:

  t = a + i*4; [t] = [t] + 1

- For safety, you must be sure that the shared expression *always* has the same value in both places!

Unsafe Common Subexpression Elimination

- Example: consider this OAT function:

```
unit f(int[] a, int[] b, int[] c) {
    int j = …; int i = …; int k = …;
    b[j] = a[i] + 1;
    c[k] = a[i];
    return;
}
```

- The optimization that shares the expression a[i] is unsafe… why?

```
unit f(int[] a, int[] b, int[] c) {
    int j = …; int i = …; int k = …;
    t = a[i];
    b[j] = t + 1;
    c[k] = t;
    return;
}
```

Code Analysis

Motivating Code Analyses

- There are lots of things that might influence the safety/applicability of an optimization
  - What algorithms and data structures can help?

- How do you know what is a loop?
- How do you know an expression is invariant (constant)?
- How do you know if an expression has no side effects?
- How do you keep track of where a variable is defined?
- How do you know where a variable is used?
- How do you know if two reference values may be aliases of one another?

Moving Towards Register Allocation

- The Oat compiler currently generates as *many* temporary variables as it needs
  - These are the %uids you should be very familiar with by now.
- Current compilation strategy:
  - Each %uid maps to a stack location.
  - This yields programs with many loads/stores to memory.
  - Very inefficient.
- Ideally, we'd like to map as many %uid's as possible into registers.
  - Eliminate the use of the alloca instruction?
  - Only 16 max registers available on 64-bit X86
  - %rsp and %rbp are reserved and some have special semantics, so really only 10 or 12 available
  - This means that a register must hold more than one slot
- When is this safe?

Liveness

- Observation: %uid1 and %uid2 can be assigned to the same register if their values will *not be needed* at the same time.
  - What does it mean for an %uid to be "needed"?
  - Ans: its contents will be used as a *source operand* in a *later* instruction.

- Such a variable is called "*live*"

- Two variables can share the same register if they are *not* live at the same time.

Scope vs. Liveness

- We can already get some coarse liveness information from variable scoping.
- Consider the following OAT program:

```
int f(int x) {
    var a = 0;
    if (x > 0) {
        var b = x * x;
        a = b + b;
    }
    var c = a * x;
    return c;
}
```

- Note that due to Oat's scoping rules, variables b and c can never be live at the same time.
  - c's scope is disjoint from b's scope
- So, we could assign b and c to the same alloca'ed slot and potentially to the same register.

But Scope is too Coarse

- Consider this program:

```
int f(int x) {
  int a = x + 2;      <- x is live
  int b = a * a;      <- a and x are live
  int c = b + x;      <- b and x are live
  return c;           <- 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

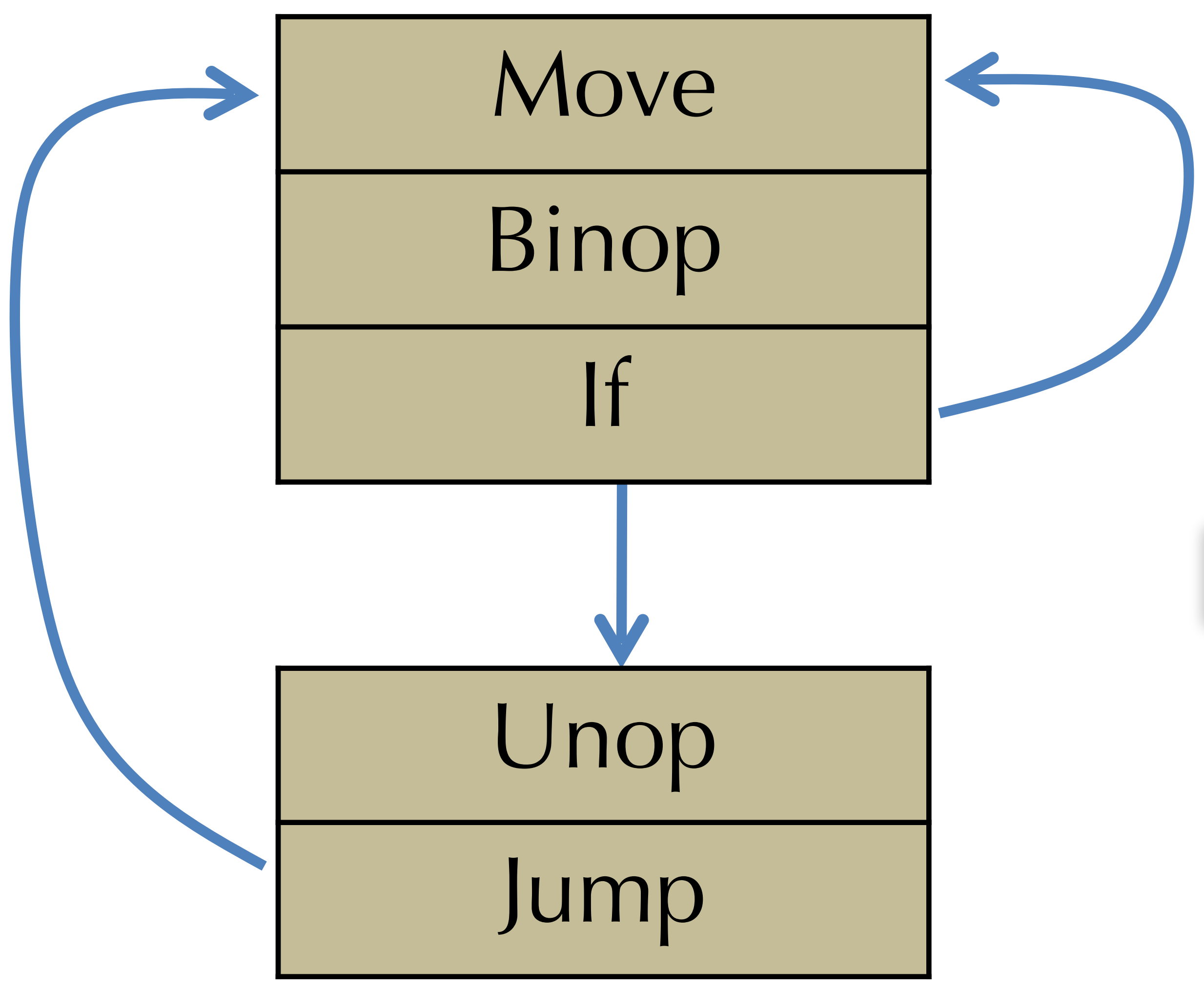
- For the purposes of dataflow analysis, we use the *control-flow graph* (CFG) intermediate form.

- 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.)

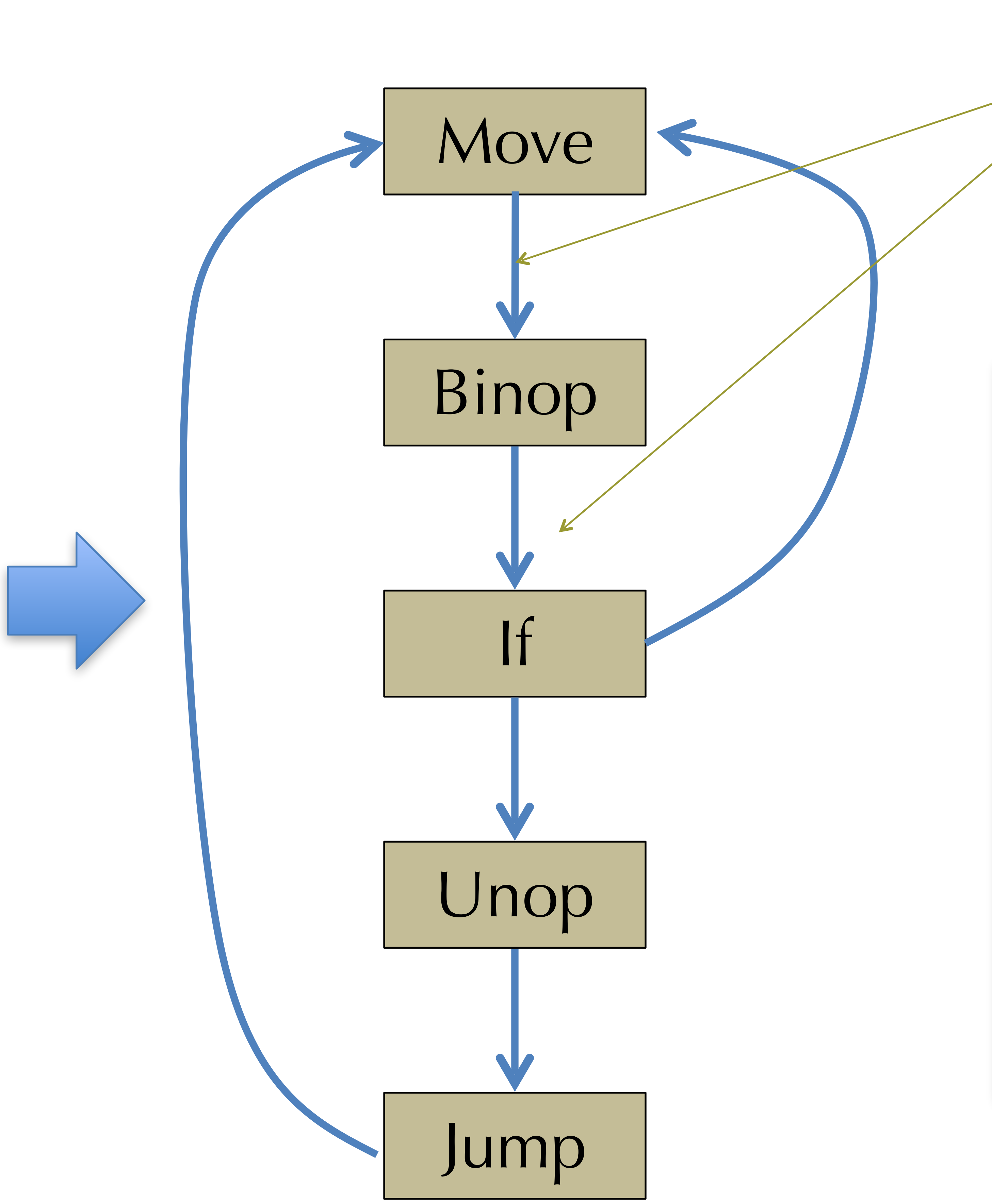
- A *control flow graph*
  - Nodes are blocks
  - There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
  - There are no “dangling” edges – there is a block for every jump target.

Dataflow over CFGs

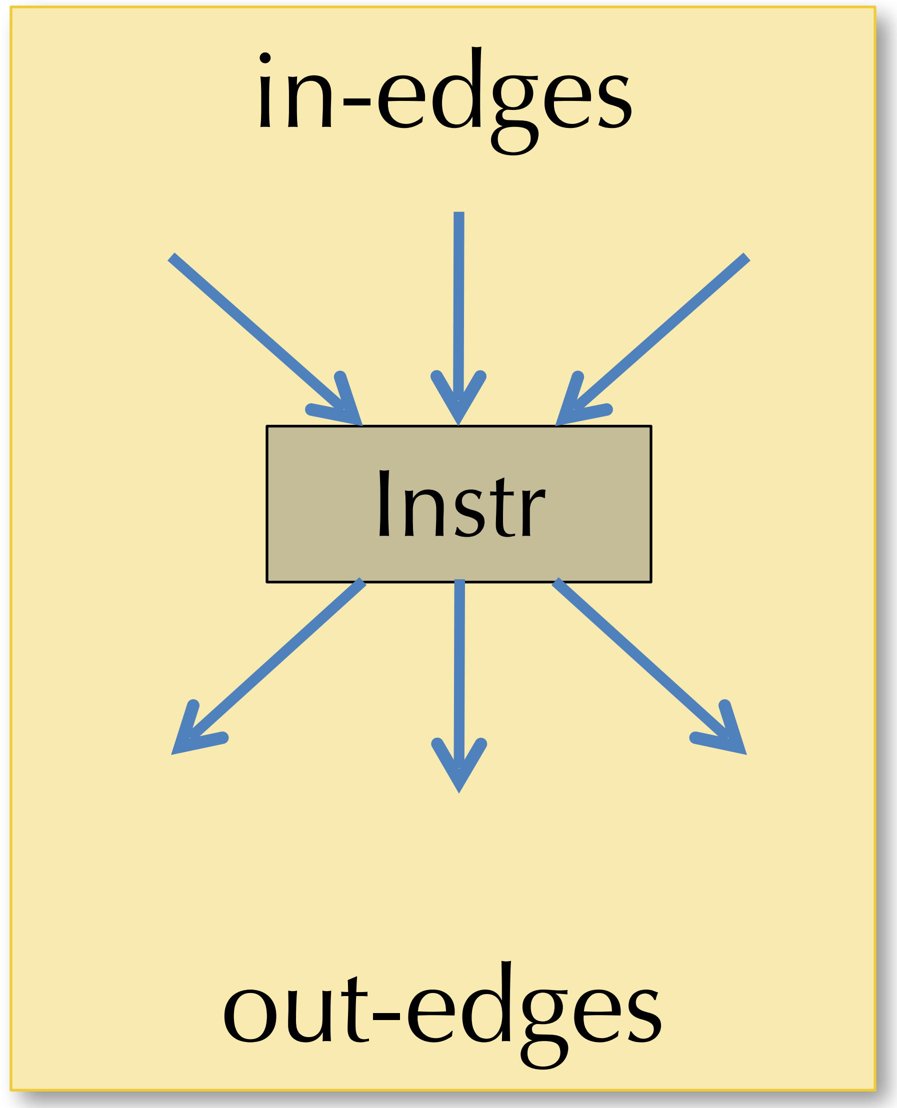
- For precision, it is helpful to think of the “fall through” between sequential instructions as an edge of the control-flow graph too.
  - Different implementation tradeoffs in practice…



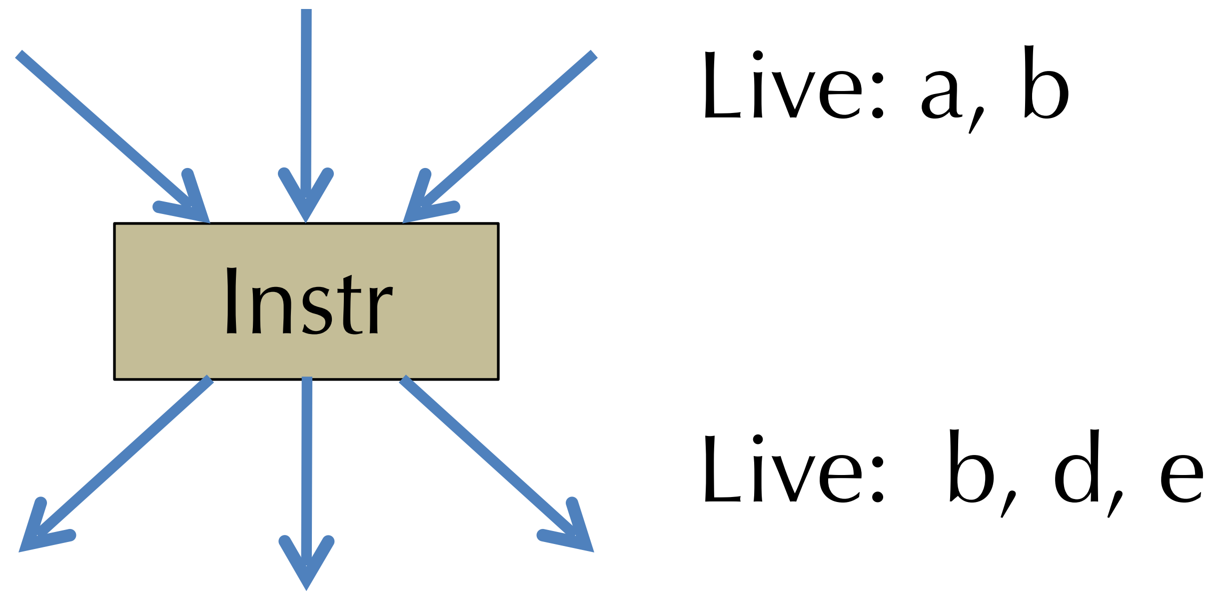
Basic block CFG



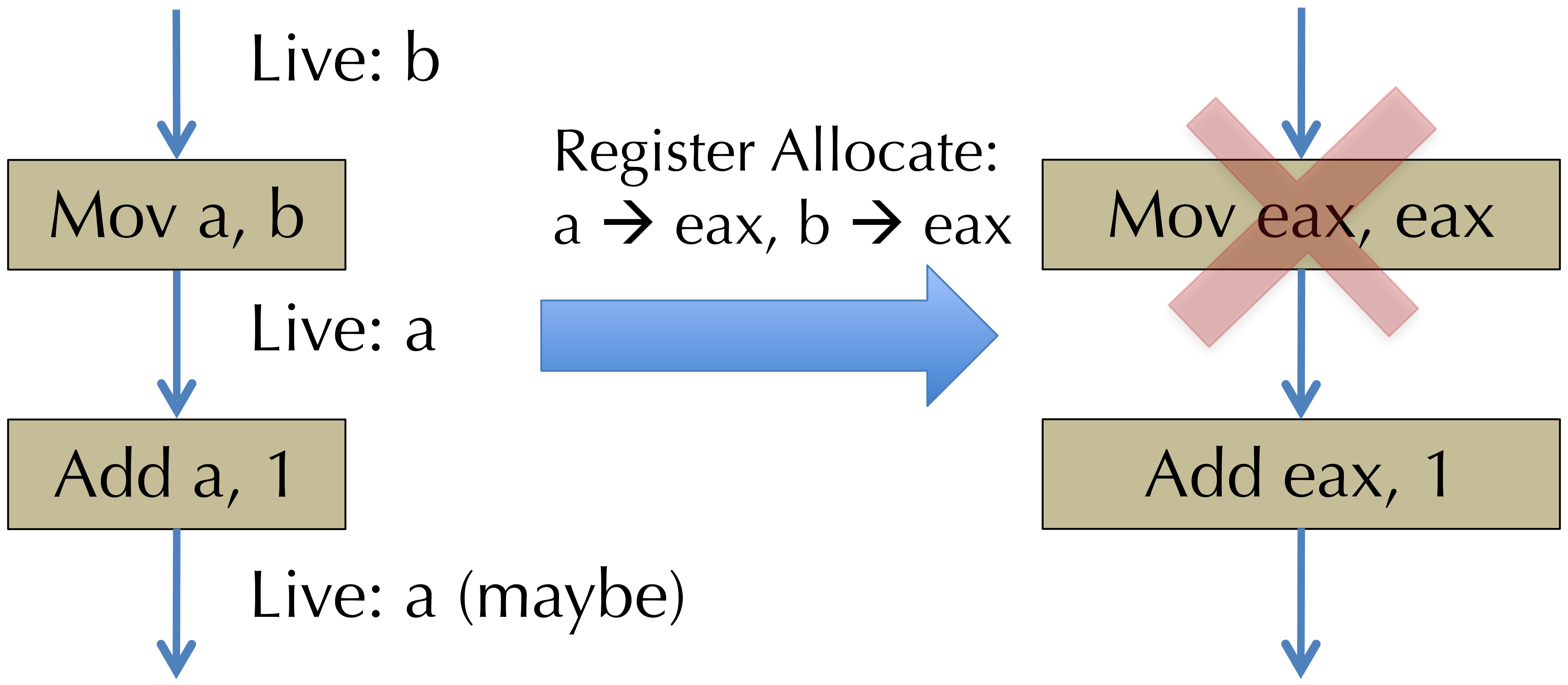
“Exploded” CFG



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 $\Rightarrow$ 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 passing 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.

- Why 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**

Time for a short break?

Dataflow Value Sets for Liveness

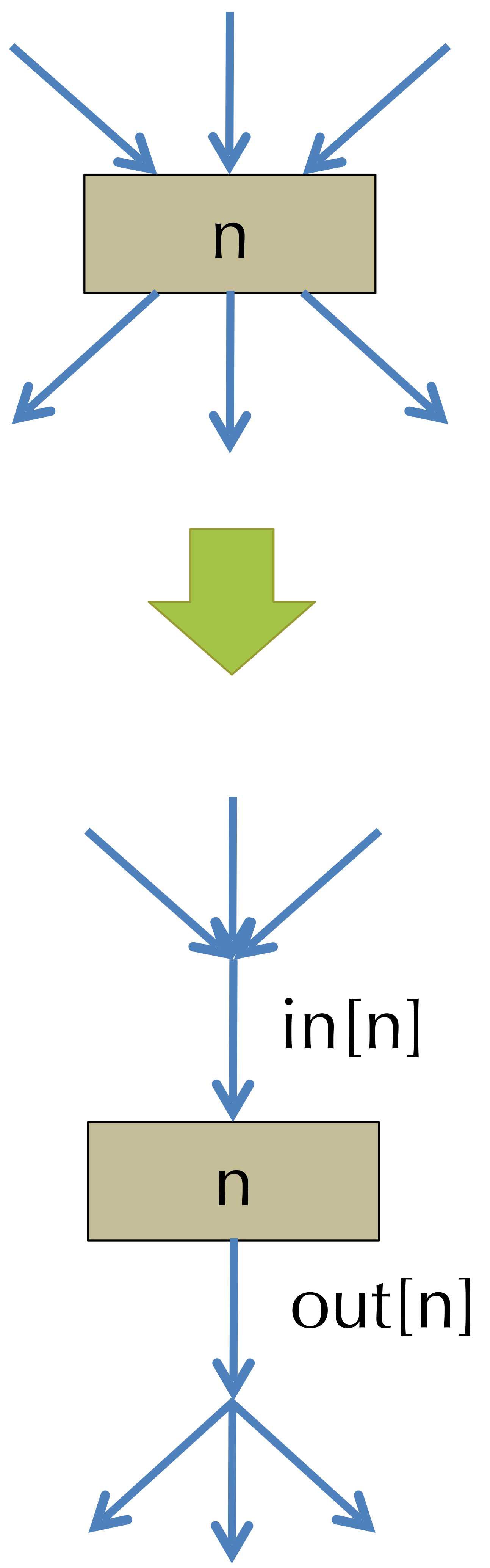
- Nodes are program statements, so:
  - use[n] : set of variables used by n
  - def[n] : set of variables defined by n
  - in[n] : set of variables live on entry to n
  - out[n] : set of variables live on exit from n

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

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

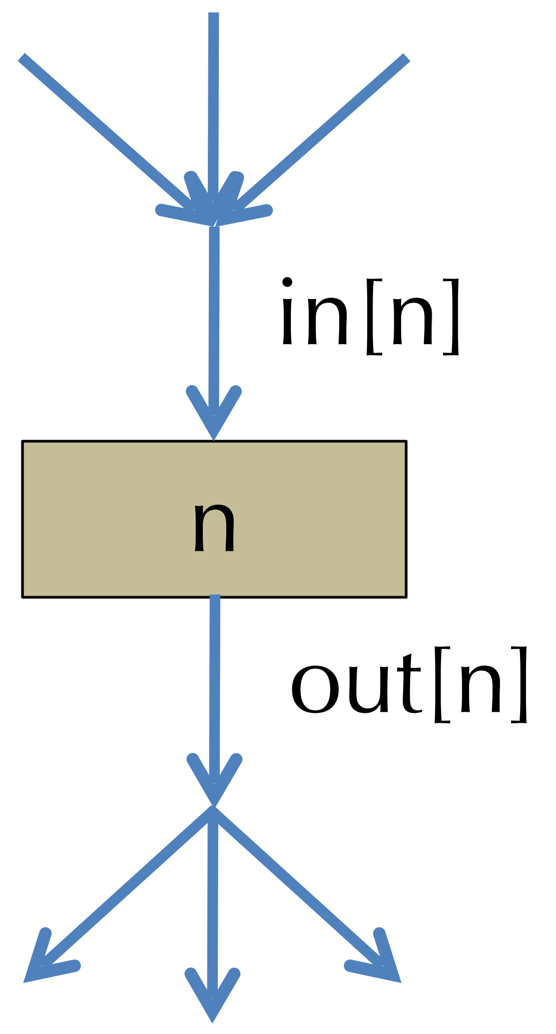
$$in[n] \supseteq use[n]$$

- What other constraints?



Other Dataflow Constraints

- We have: $in[n] \supseteq use[n]$
  - "A variable must be live on entry to n if it is used by n"
- Also: $in[n] \supseteq out[n] - def[n]$
  - "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: $out[n] \supseteq in[n']$ if $n' \in succ[n]$
  - "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) What are they?

  - $in[n] := use[n] \cup (out[n] - def[n])$
  - $out[n] := \bigcup_{n' \in succ[n]} in[n']$

Complete Liveness Analysis Algorithm

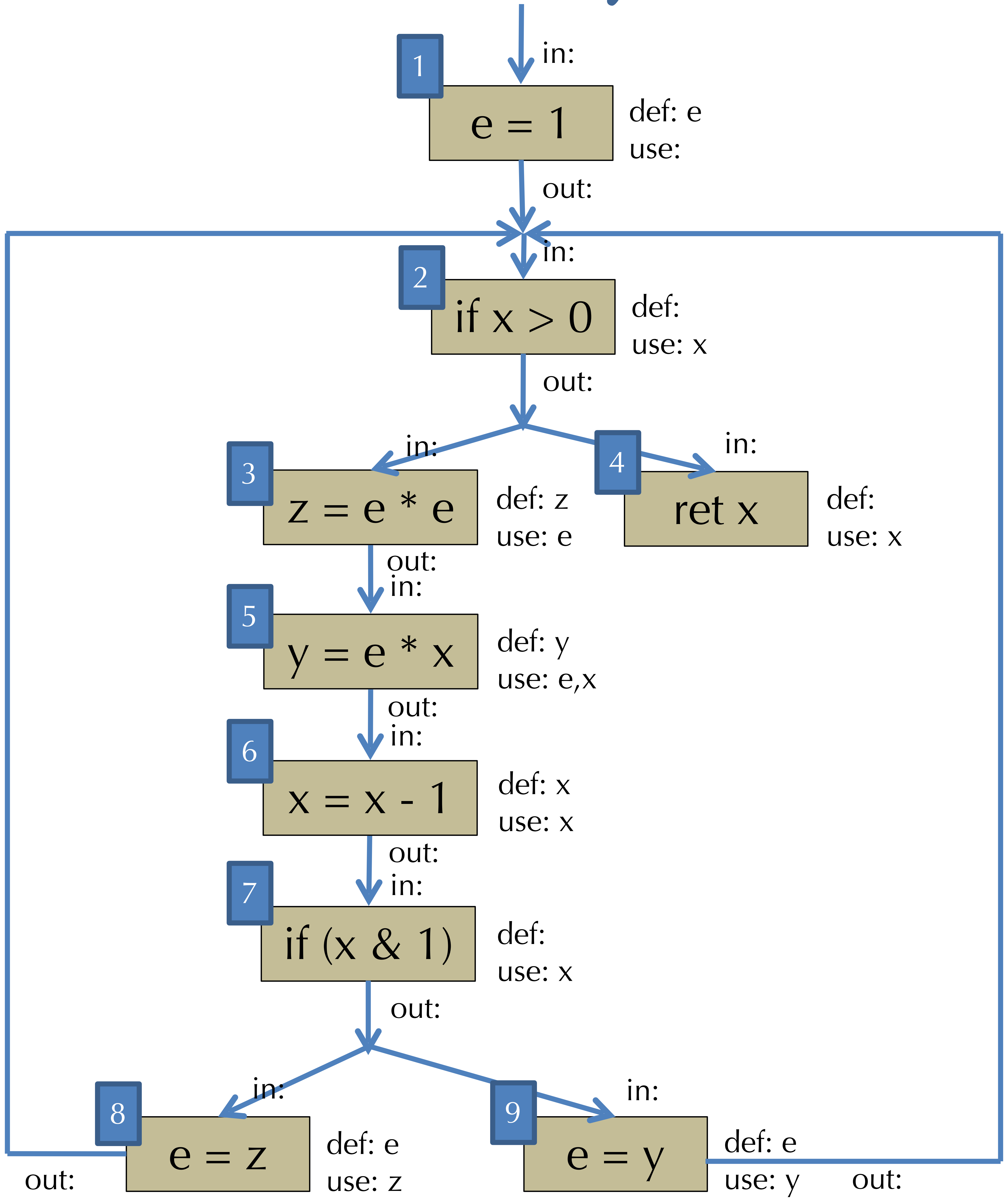
for all n, in[n] := Ø, out[n] := Ø
repeat until no change in 'in' and 'out'
  for all n
    $out[n] := \bigcup_{n' \in succ[n]} in[n']$
    $in[n] := use[n] \cup (out[n] - def[n])$
  end
end

- 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

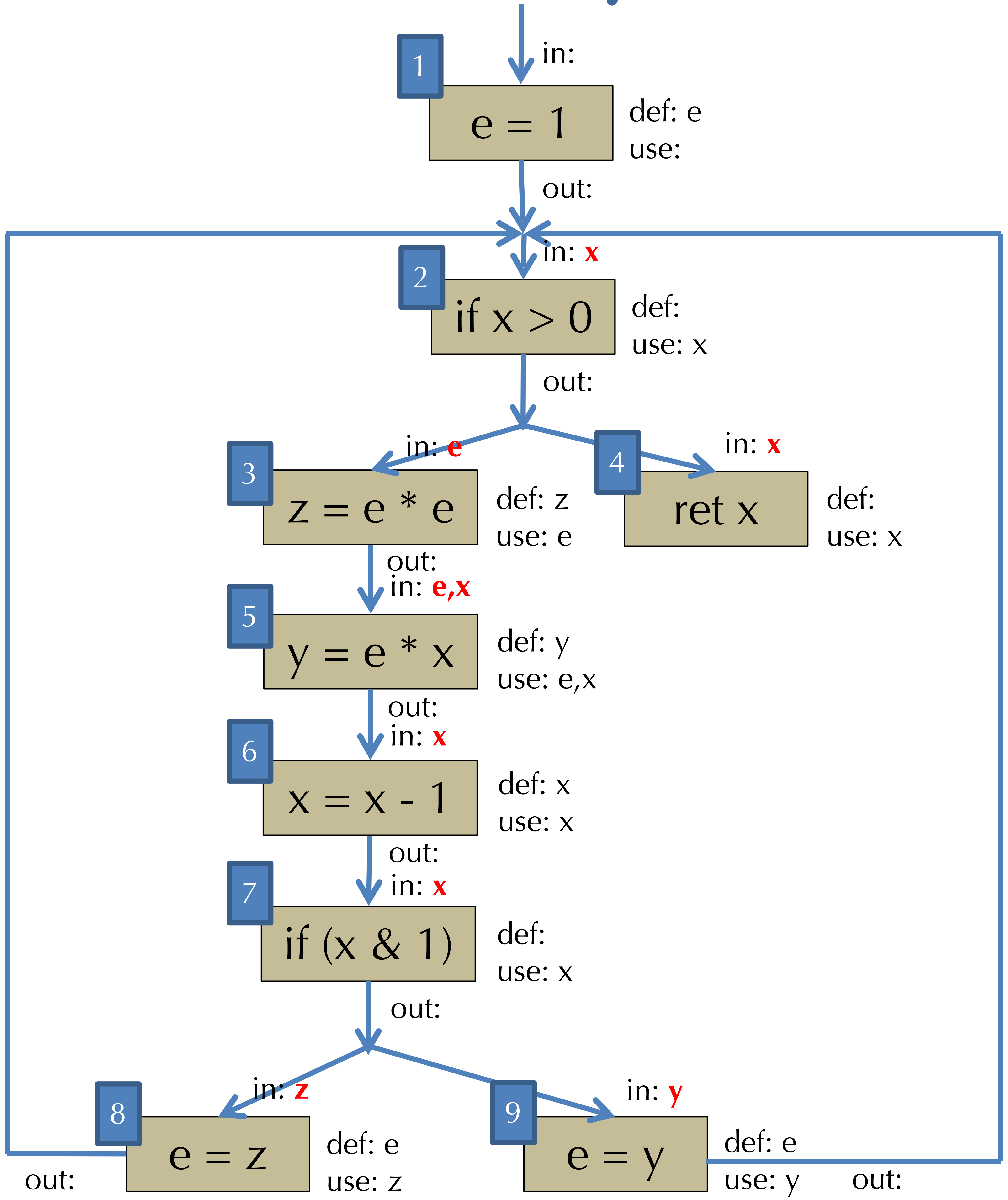
in[6] = x

in[7] = x

in[8] = z

in[9] = y

(showing only updates that make a change)



Example Liveness Analysis

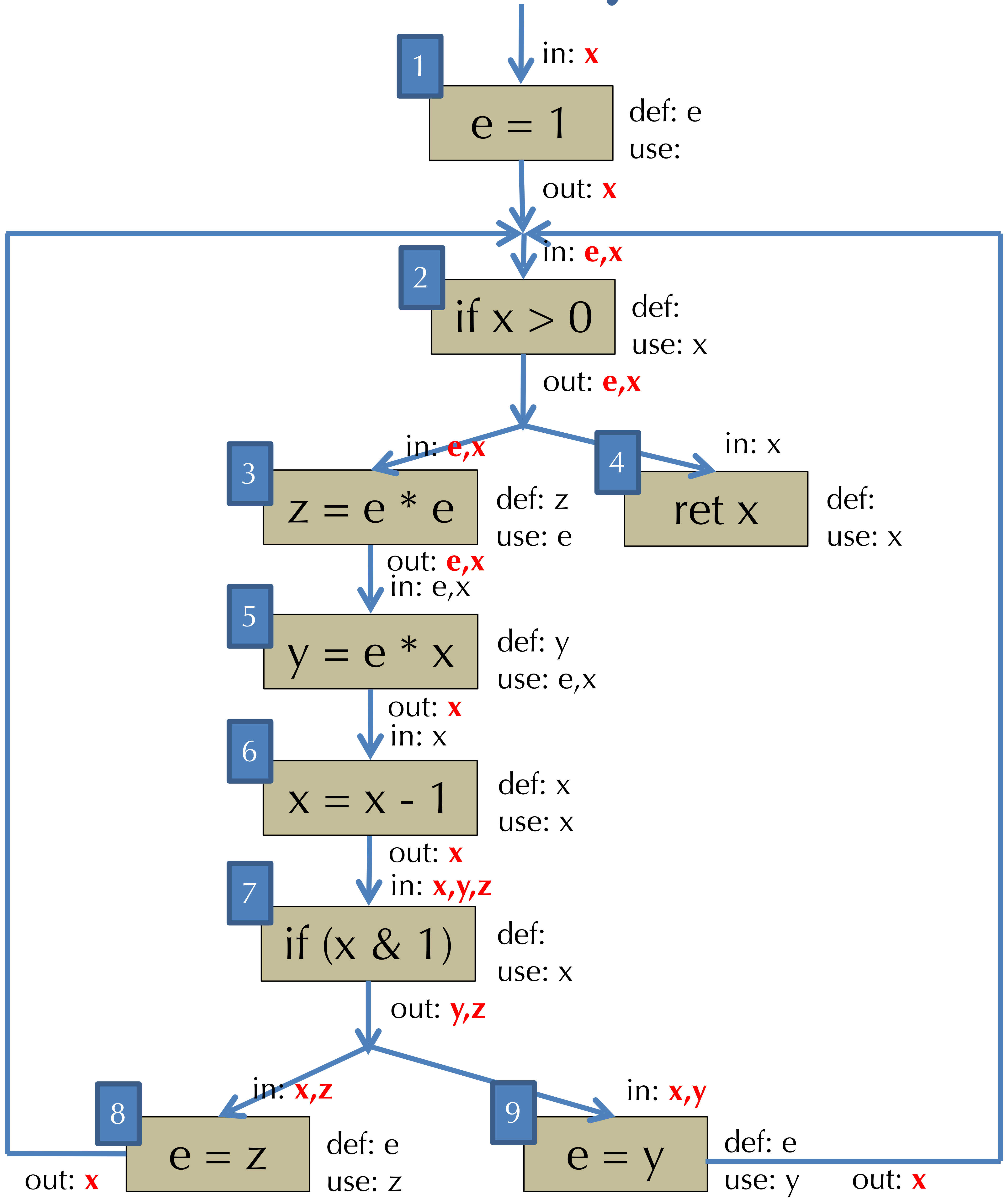
Each iteration update:

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

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

- 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

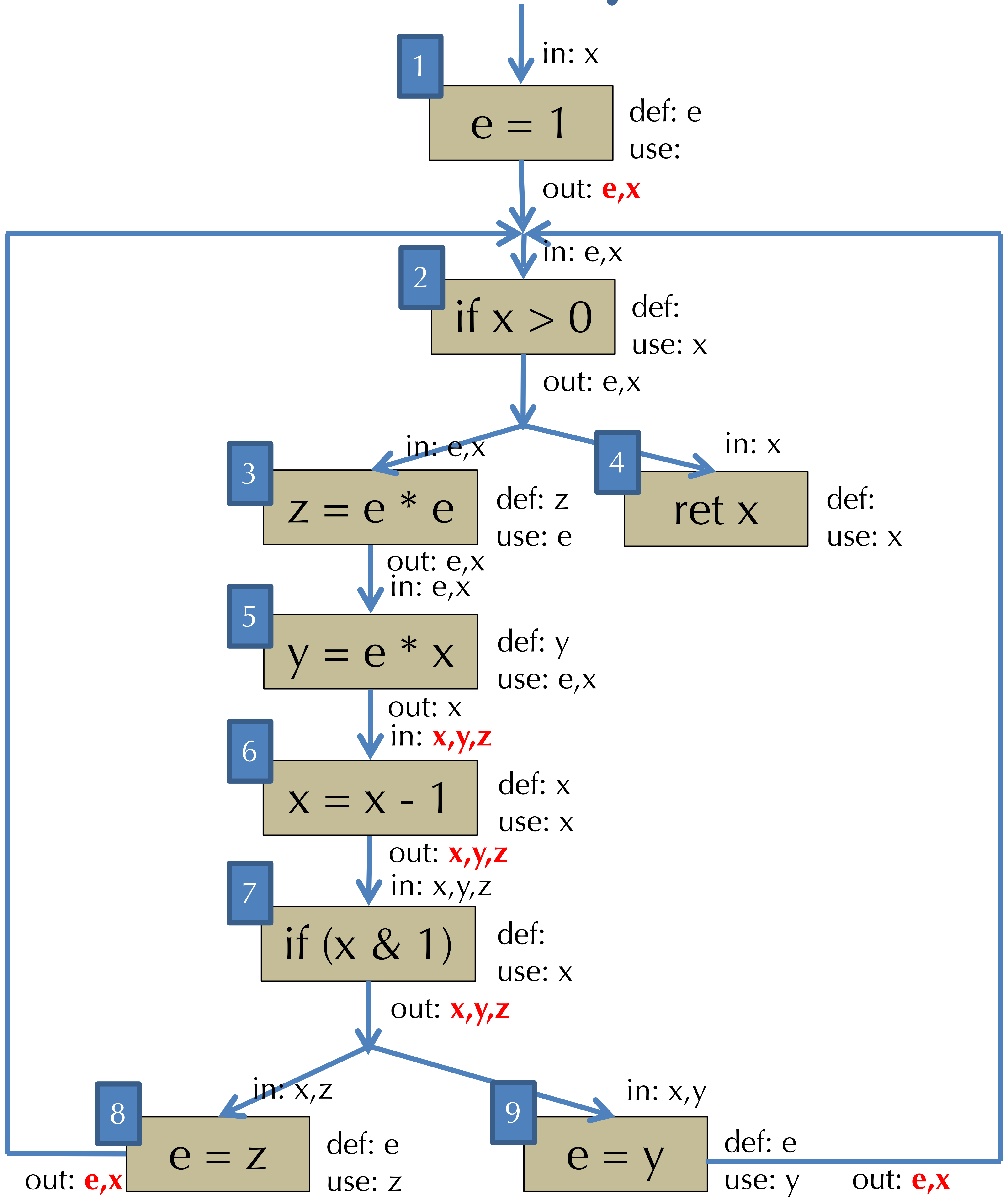
out[6]= x,y,z

in[6]= x,y,z

out[7]= x,y,z

out[8]= e,x

out[9]= e,x



Example Liveness Analysis

Each iteration update:

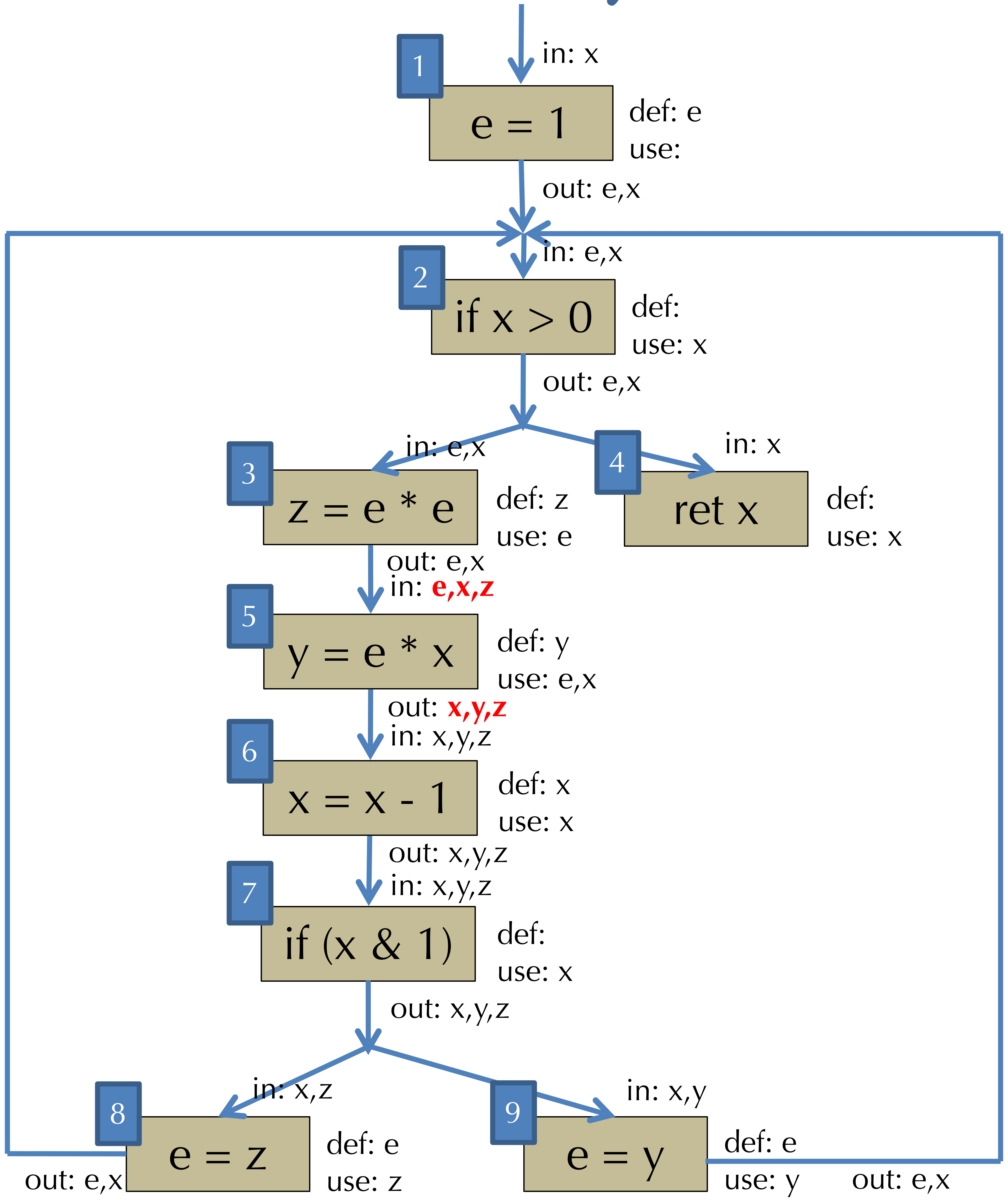
$out[n] := \cup_{n' \in succ[n]} in[n']$

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

- Iteration 4:

out[5]= x,y,z

in[5]= e,x,z



Example Liveness Analysis

Each iteration update:

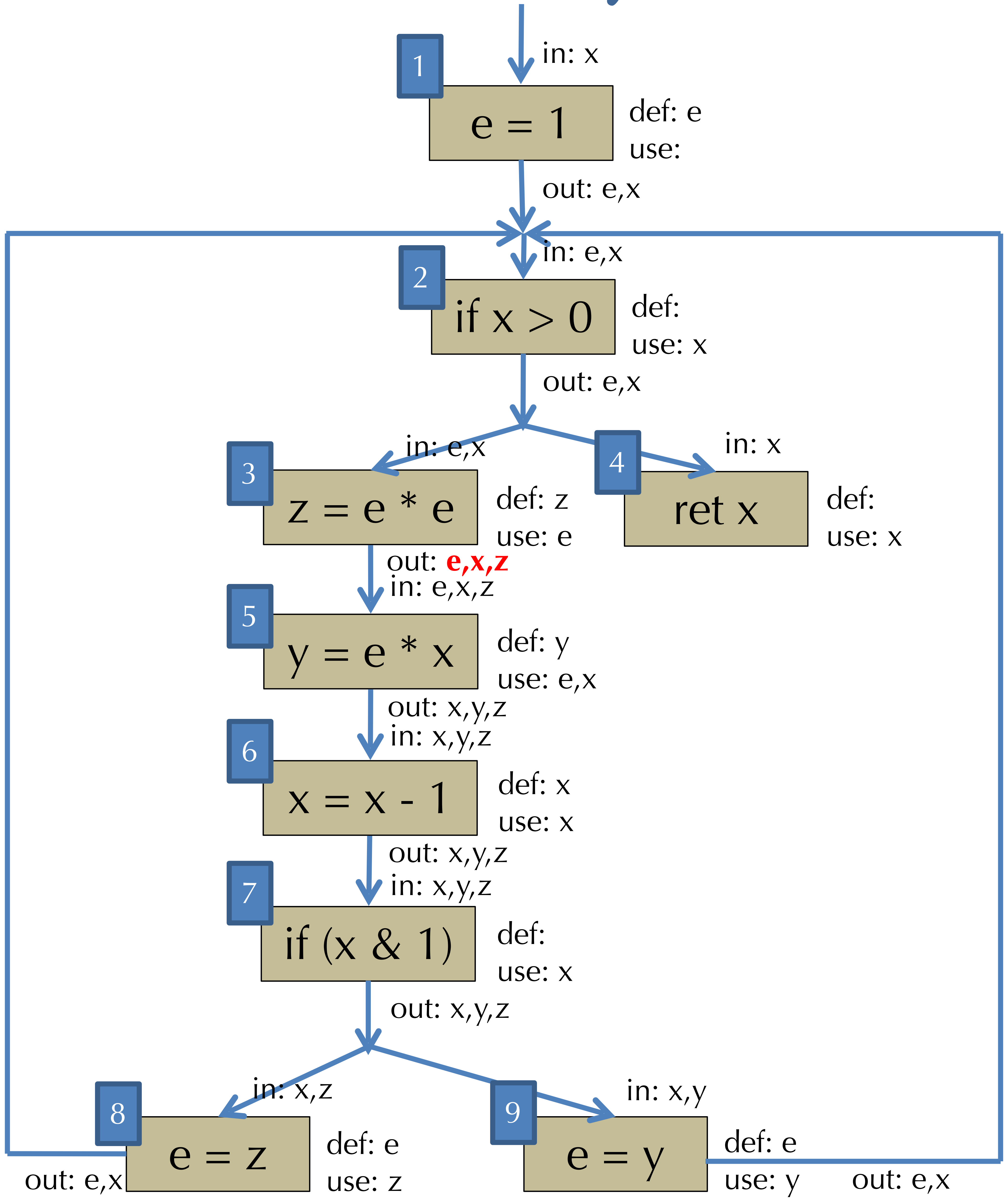
$out[n] := \bigcup_{n' \in succ[n]} in[n']$

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

- Iteration 5:

out[3]= e,x,z

Done!



Improving the Iterative Algorithm

- Can we do better?

- Observe: the only way information propagates from one node to another is using: $\mathrm{out}[n] := \bigcup_{n' \in \mathrm{succ}[n]} \mathrm{in}[n']$
  - 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

Generalising Dataflow Analyses

- The kind of iterative constraint solving used for liveness applies to other kinds of analyses.
  - Reaching Definitions analysis
  - Available Expressions analysis
  - Alias Analysis
  - Constant Propagation
  - These analyses follow the same approach as for liveness (accumulating values until fixpoint is reached).

- 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* (op, arg1, arg2, and result)
  - 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

Reaching Definitions

Reaching Definition Analysis

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

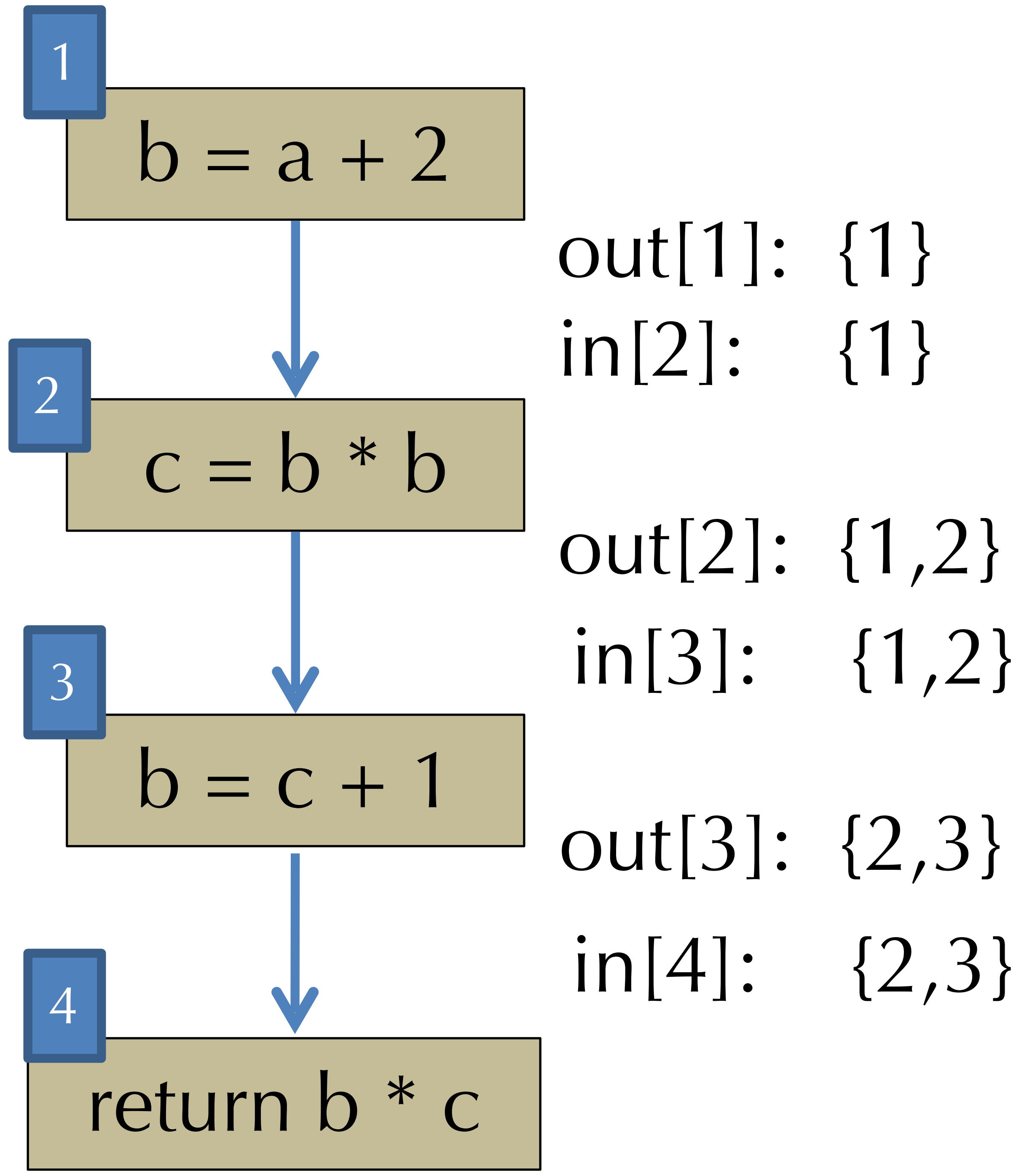
- Unlike liveness, we are interested in *different* definitions of the same variable.

- This analysis is used for constant propagation & copy propagation
  - 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* (statements) 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

- gen[n] are node's definitions; kill[n] are the nodes, whose definitions are "shadowed" by n

Reaching Definitions Step 2

- Define the constraints that a reaching definitions solution must satisfy.

- $out[n] \supseteq gen[n]$
  - "The definitions that reach the end of a node at least include the definitions generated by the node"

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

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

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: initialise in[n] and out[n] to Ø
  – Iterate the update equations until a fixed point is reached
  – Why does it terminate?

- The algorithm terminates because in[n] and out[n] increase only *monotonically*
  – At most to a maximum set that includes all variable definitions 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          a = x + 1
    ...         ⇒      ...
    b = x + 1          b = a
    ```
  - When is it safe?

- 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
  - Let uses[a] be the set of *nodes* that use the variable a in their expressions
- 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	Ø	uses[[x]] (for all x that may equal a)
br L	Ø	Ø
br a L1 L2	Ø	Ø
$a = f(b_1,\ldots,b_n)$	Ø	uses[a] ∪ uses[[x]] (for all x)
$f(b_1,\ldots,b_n)$	Ø	uses[[x]] (for all x)
return a	Ø	Ø

Note the need for "may alias" information…

Note that functions are assumed to be impure…

  - gen[n] — node itself represents new available expression
  - kill[n] — nodes whose expressions no longer available after n

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:
  - $in[n] := \bigcap_{n' \in pred[n]} out[n']$
  - $out[n] := gen[n] \cup (in[n] - kill[n])$

- Unlike previous algorithms, this one is "shrinking" the set of desired facts

- Algorithm: initialise in[n] and out[n] to {set of all nodes}
  - Iterate the update equations until a fixed point is reached
  - Why does the algorithm terminate?

- 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.

How about another break?

General Dataflow Analysis Framework

Comparing Dataflow Analyses

- Look at the update equations in the inner loop of the analyses

- Liveness:
  - Let gen[n] = use[n] and kill[n] = def[n]
  - $\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$ (backward)
  - $\text{in}[n] := \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])$

- Reaching Definitions:
  - $\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n']$ (forward)
  - $\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$

- Available Expressions:
  - $\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']$ (forward)
  - $\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$

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: what is added, what is removed.

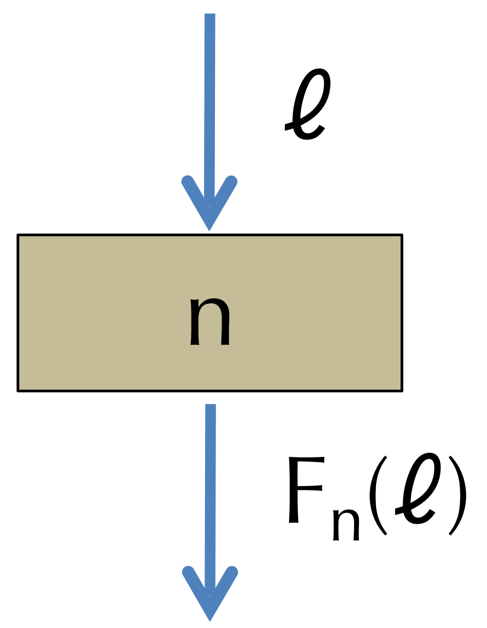
- Each analysis is propagates information either *forward* or *backward*
  - Forward: in[n] defined in terms of predecessor nodes' out[]
  - Backward: out[n] defined in terms of successor nodes' in[]

- Each analysis has a way of aggregating (combining) information from in/out flow
  - Liveness & reaching definitions take union ($\cup$)
  - Available expressions uses intersection ($\cap$)
  - Union expresses a property that holds for *some* path (existential)
  - Intersection expresses a property that holds for *all* paths (universal)

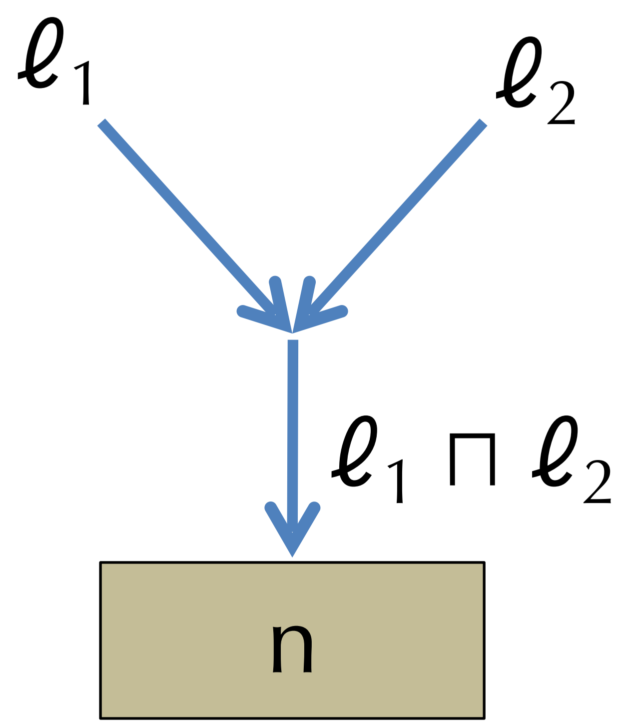
(Forward) Dataflow Analysis Framework

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 "$z \in \ell$" means "z has the property"
2. For each node n, a flow function $F_n : \mathcal{L} \to \mathcal{L}$
   - 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 $\sqcap$
   - "If we know *either* ℓ_1 *or* ℓ_2 holds on entry to node n, we know at most $\ell_1 \sqcap \ell_2$"
   - $in[n] := \bigsqcap_{n' \in pred[n]} out[n']$



Generic Iterative (Forward) Analysis

for all n, in[n] := $\top$, out[n] := $\top$

repeat until no change

    for all n

        in[n] := $\sqcap_{n' \in pred[n]}$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, so we can't keep it. :-(
  - Iteration refines the answer, eliminating inconsistencies

Structure of $\mathcal{L}$

- The domain has structure that reflects the "amount" of information for 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 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 $\ell_1 \subseteq \ell_2$

- Example 2: 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 $\ell_1 \supseteq \ell_2$

$\mathcal{L}$ 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 $\supseteq$

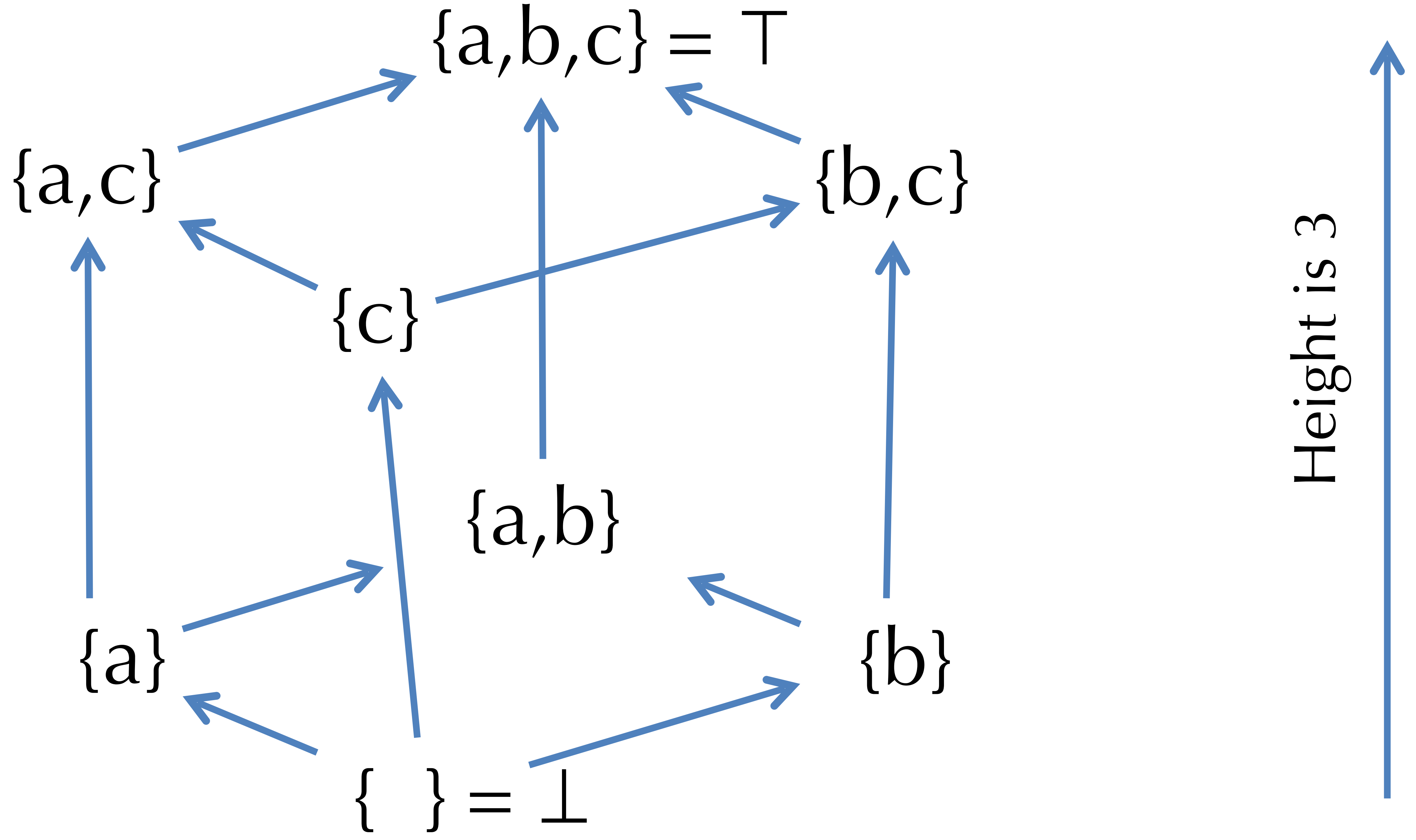
Subsets of {a,b,c} ordered by ⊆

ℓ_2

$\ell_1 \sqsubseteq \ell_2$

ℓ_1

Partial orders are often presented as a Hasse diagram.



order $\sqsubseteq$ is $\subseteq$ meet $\sqcap$ is $\cap$ join $\sqcup$ is $\cup$

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 (Forward) Algorithm

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

- Equivalently: $out[n] := F_n(\sqcap_{n' \in pred[n]} out[n'])$
  - 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 $\mathcal{L}$ corresponding to CFG nodes
  - $\mathbf{F}(\mathbf{X}) = (F_1(\sqcap_{j \in pred[1]} out[j]), F_2(\sqcap_{j \in pred[2]} out[j]), \ldots, F_n(\sqcap_{j \in pred[n]} out[j]))$

- Any solution to the constraints is a *fixpoint* $\mathbf{X}$ of $\mathbf{F}$
  - 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

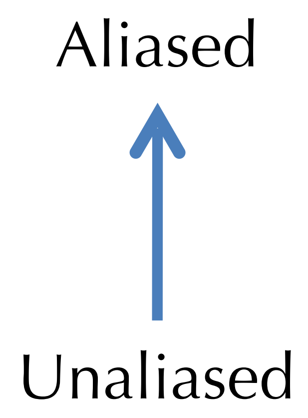
- Note that **F** is consistent: $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\ldots \sqsubseteq \mathbf{F}(\mathbf{F}(\mathbf{X}_0)) \sqsubseteq \mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$

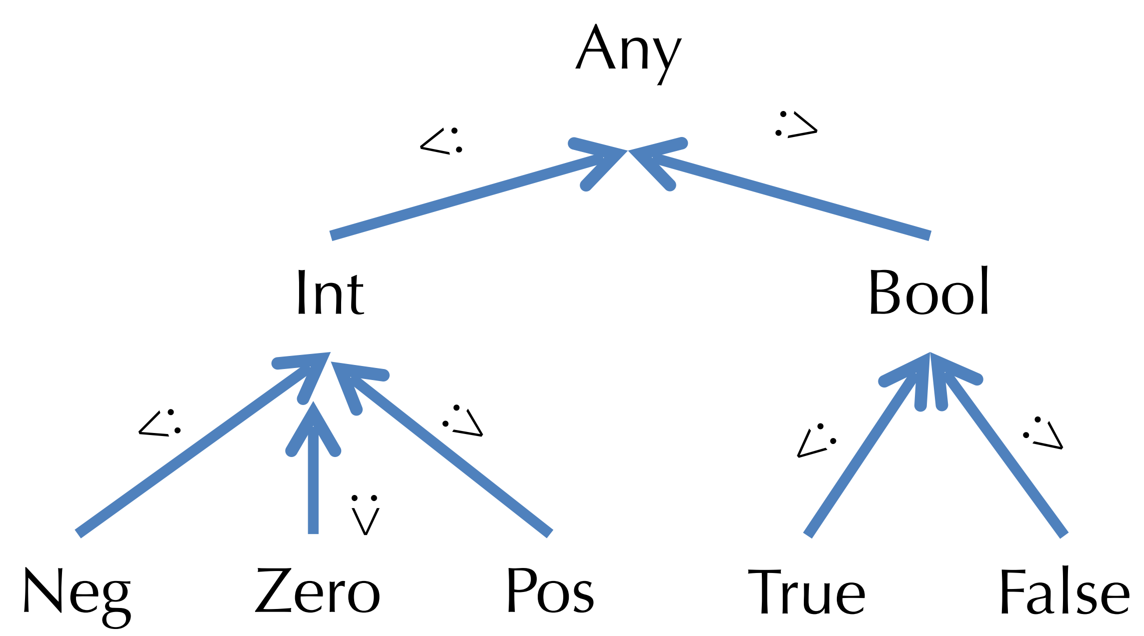
- Therefore, # steps needed to reach a fixpoint is at most the height H of $\mathcal{L}$ times the number of nodes: $O(H_n)$ — height of the lattice

Building Lattices?

- Information about individual nodes or variables can be lifted *pointwise:*
  - If $\mathcal{L}$ is a lattice, then so is $\{ f : X \to \mathcal{L} \}$ 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*"

More on Fixpoint Solutions

- Remember constructing LL(1) parse tables

$T \longmapsto S\$$
$S \longmapsto ES'$
$S' \longmapsto \varepsilon$
$S' \longmapsto + S$
$E \longmapsto \text{number} \mid (S)$

- First(T) = First(S)
- First(S) = First(E)
- First(S′) = { + }
- First(E) = { number, '(' }

- Follow(S′) = Follow(S)
- Follow(S) = { \$, ')' } ∪ Follow(S′)

Then: we want the *least* solution to this system of set equations… a *fixpoint* computation. More on these later in the course.

Now: This solution is obtained by starting from taking all First/Follow as ∅ and then iterating the equations until *fixpoint* is reached.

	number	+	(	)	\$ (EOF)
T	$\longmapsto$ S\$		$\longmapsto$S\$		
S	$\longmapsto$ E S′		$\longmapsto$E S′		
S′		$\longmapsto$ + S		$\longmapsto$ ε	$\longmapsto$ ε
E	$\longmapsto$ num.		$\longmapsto$ (S)		

Dataflow Analysis: Summary

- Many dataflow analyses fit into a common framework.
- Key idea: *iterative solution* of a system of equations over a *lattice* of *facts* (constraints).
  - Iteration terminates if flow functions are *monotonic*.
  - Solution is obtained as the *greatest* fixpoint is reached via the *meet* operation ($\sqcap$).
- In the literature, sometimes the definition of the analysis lattice is *reversed*:
  - The most useful/precise information is represented by the bottom element ($\perp$)
  - Solution is obtained as the *least* fixpoint via iterative application of *join* operator ($\sqcup$)
  - The two definitions are equivalent modulo the (semi-)lattice *direction*.

Next Lecture (Finally!)

- Register Allocation
- Modern research directions in PLDI
- Wrap-Up