Lecture 22

CIS 341: COMPILERS

Announcements

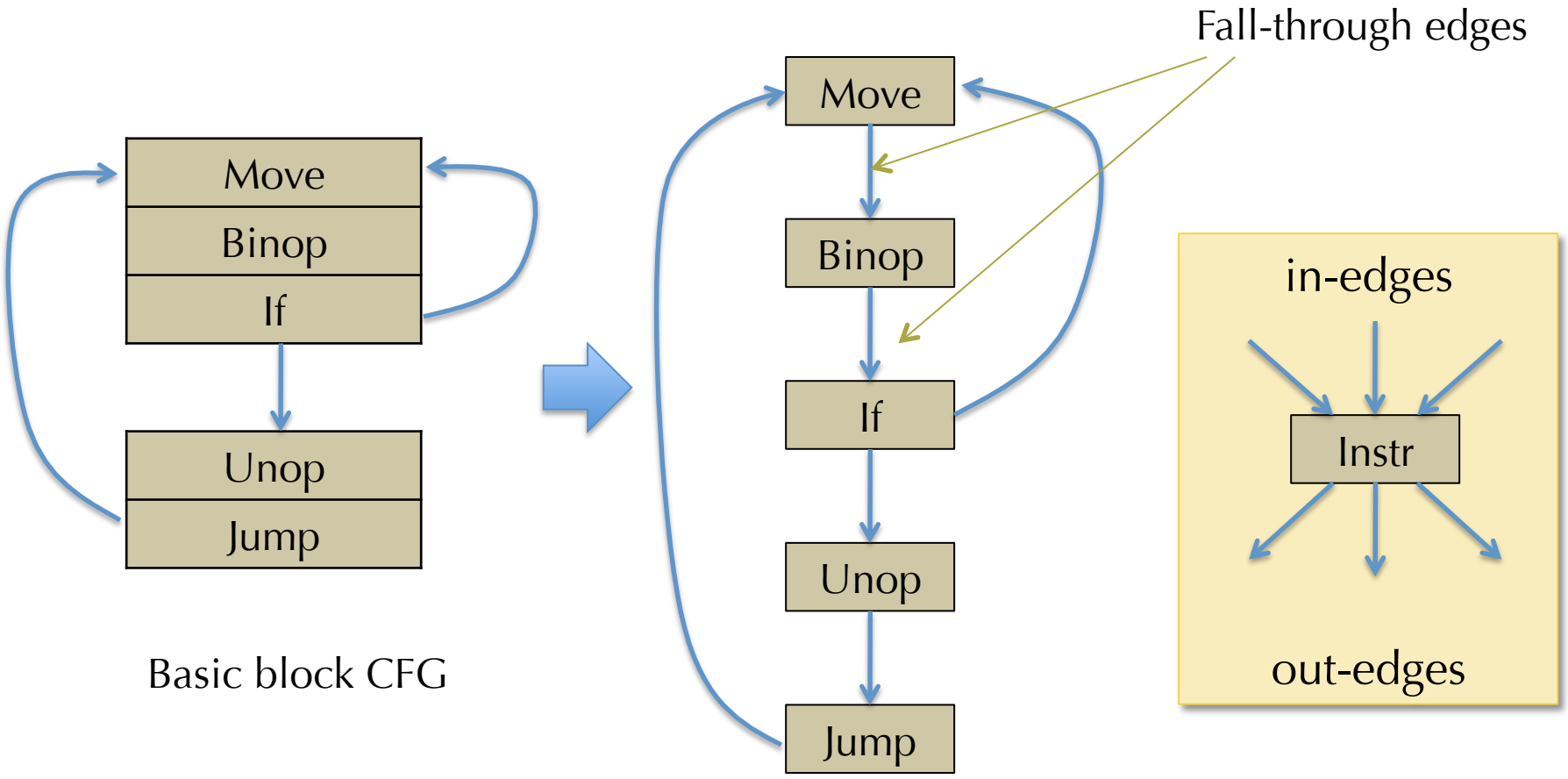
- Projects 6 & 7
  - Information available soon (after late deadline for Project 5)
  - Should be significantly lighter work load than Project 5

- Final Exam:
  - Tuesday, April 30th noon-2:00 pm
  - Moore 216

CODE ANALYSIS

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.
  - In practice, identify instructions by offsets within basic blocks



Example Dataflow Analyses

- Liveness:
  - Computes, for each edge, the set of variables that are live across the edge.
  - A variable is live across the edge if it used before it is redefined on the path originating at the edge.

- Reaching Definitions:
  - Computes, for each edge, the set of variables whose definitions might reach the edge.
  - Useful for constant propagation. (If there is only one reaching definition to a node and it's a constant, it's OK to substitute the constant for uses.)

- Available Expressions:
  - Computes, for each edge, the set of expressions whose values are the same along all paths to the edge.

A Worklist Algorithm for Livemeness

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

```
for all n, in[n] := ∅, out[n] := ∅
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
```

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 = [b]	{n} - kill[n]	uses[a]
[a] = b	Ø	uses[[x]] (for all x that may equal a)
jump L	Ø	Ø
if a goto L1 else L2	Ø	Ø
L:	Ø	Ø
a = f(b_1,…,b_n)	Ø	uses[a] ∪ uses[[x]] (for all x)
f(b_1,…,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…

Available Expressions Step 2

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

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

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

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

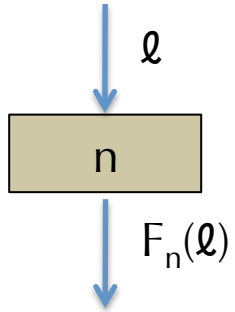
Common Features

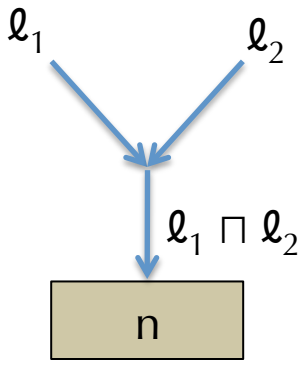
- All of these analyses have a *domain* over which they solve constraints.
  - Liveness, the domain is sets of variables
  - Reaching defns., Available exprs. the domain is sets of nodes
- Each analysis has a notion of gen[n] and kill[n]
  - Used to explain how information propagates across a node.
- 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 ($\cup$)
  - 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 "x $\in \ell$" 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 $\sqcap$
   - "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

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

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

    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

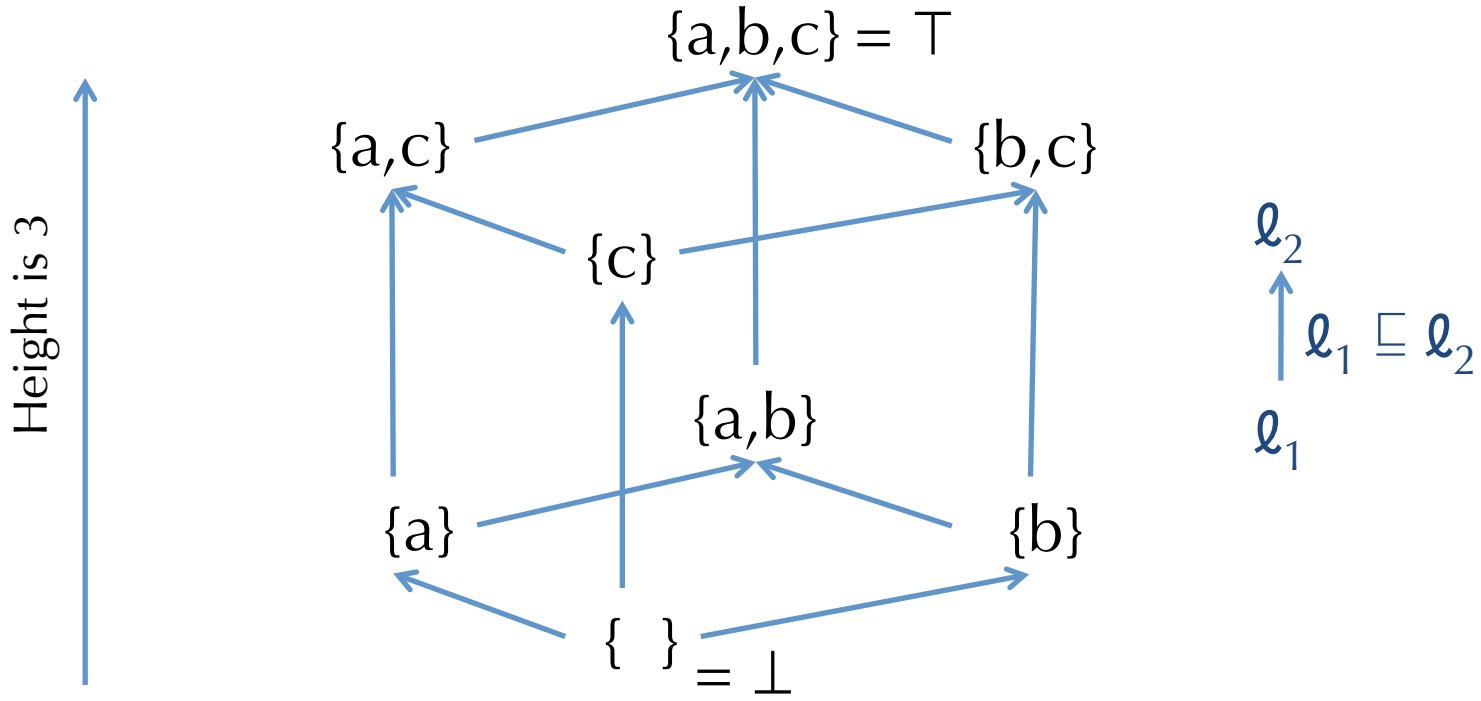
- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq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

- 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 precise information entering a node, then you have more precise 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)

Reaching Definitions is Monotone

- Reaching definitions is a forward analysis.
- $in[n] = \bigcup_{n' \in pred[n]} out[n']$
- $out[n] = gen[n] \cup (in[n] - kill[n])$
- Top $\top = \emptyset$, meet $\sqcap = \cup$, join $\sqcup$ is $\cap$, order $\sqsubseteq$ is $\supseteq$
- So: $F_n[x] = gen[n] \cup (x - kill[n])$
  is it monotonic? Let $x \sqsubseteq z$, so $z = x \sqcup y$ for some y.
- $F_n[x \sqcup y]$
  $= gen[n] \cup ((x \cap y) - kill[n])$
  $= gen[n] \cup ((x - kill[n]) \cap (y - kill[n]))$
  $= (gen[n] \cup (x - kill[n])) \cap (gen[n] \cup (y - kill[n]))$
  $= F_n[x] \cap F_n[y]$
  $\subseteq F_n[x]$

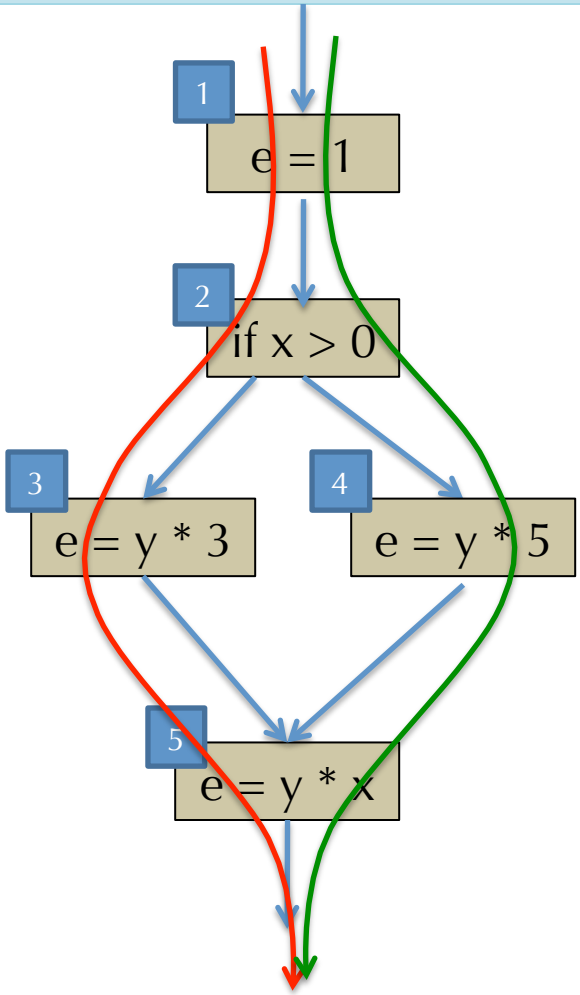
QUALITY OF DATAFLOW ANALYSIS SOLUTIONS

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:

  $\bigsqcap_{p \in paths\_to[n]} \ell_p$



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.

"Classic" Constant Propagation

- Constant propagation can be formulated as a dataflow analysis.

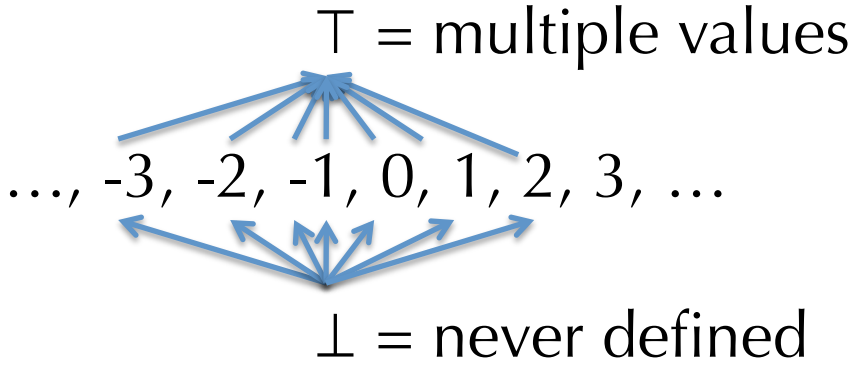
- Idea: propagate and fold integer constants in one pass:

```
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."

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

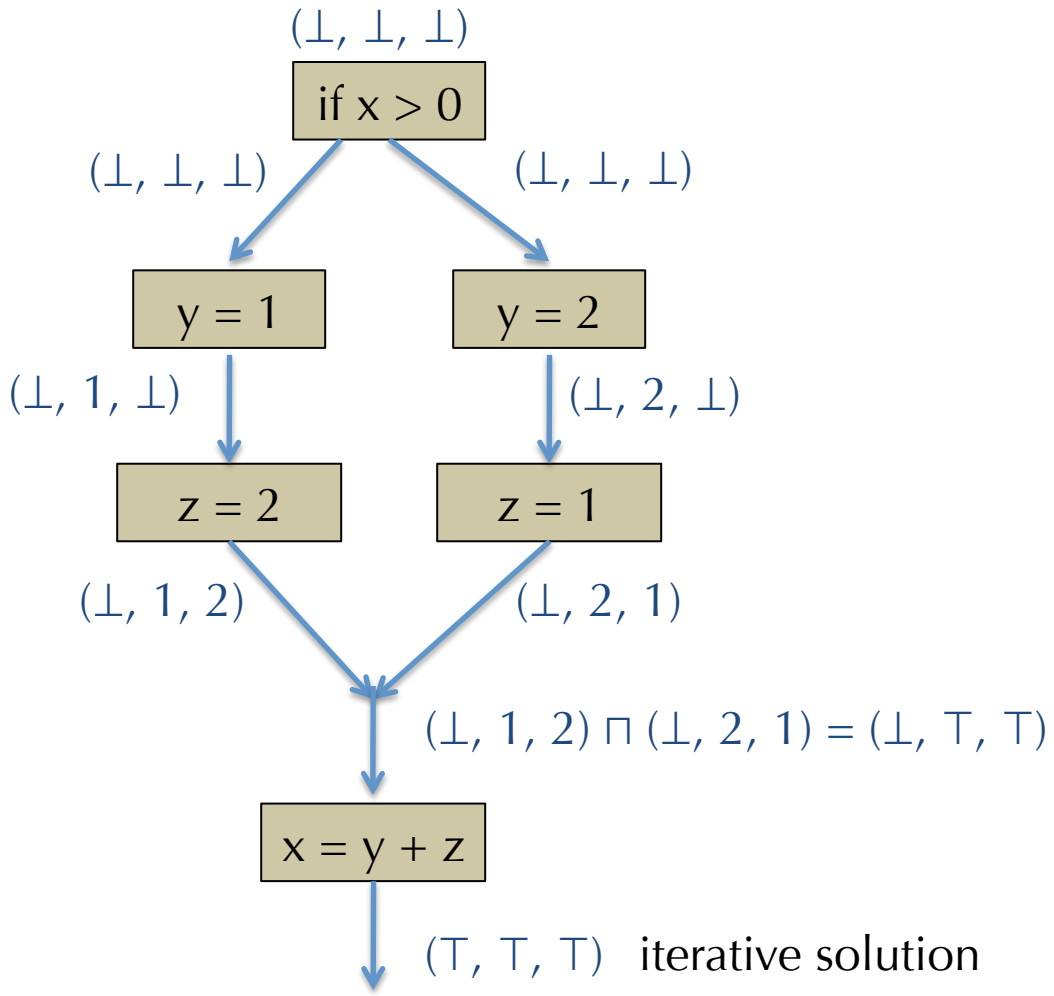
"If either input is undefined the result of the operation is too."

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

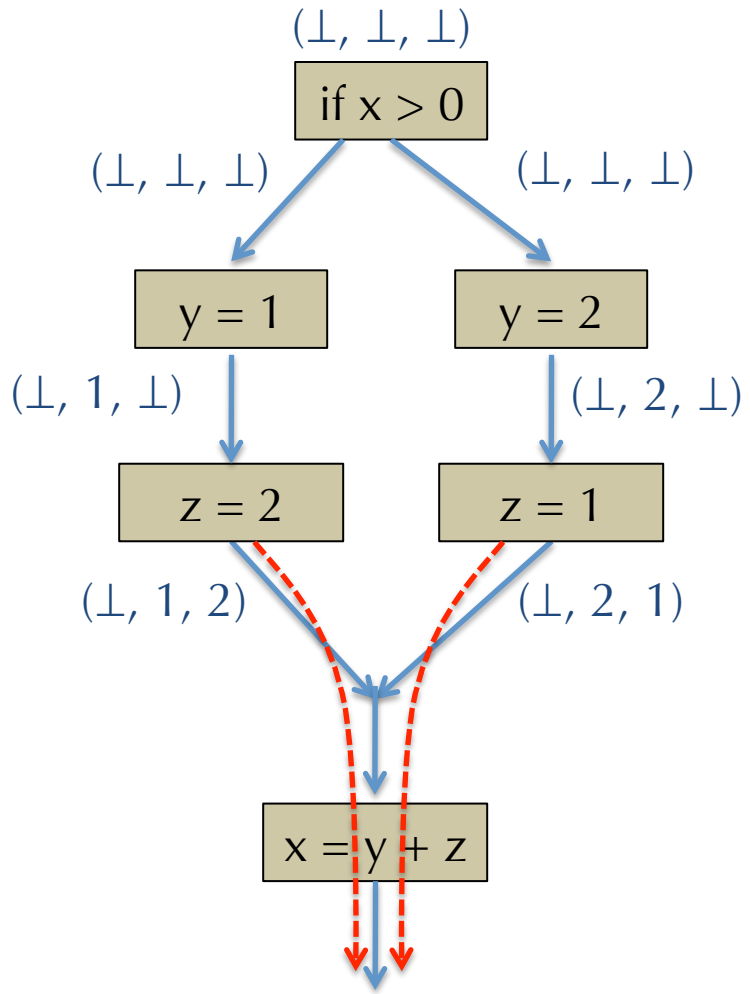
"If the inputs are known constants, calculate the output statically."

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

Iterative Solution



MOP Solution ≠ Iterative Solution



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

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