Lecture 24

CS131: COMPILERS

Announcements

- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Due: December 30th

- Final Exam:
  - In class, Jan 2nd
  - Coverage: emphasizes material since the midterm
  - Cheat sheet: one, hand-written, double-sided, letter-sized page of notes

GENERAL DATAFLOW ANALYSIS

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
```

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- out[n] := $\bigcup_{n' \in succ[n]}$ in[n'] (backward)
- in[n] := gen[n] ∪ (out[n] - kill[n])

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

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

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

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

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- $out[n] := \bigcup_{n' \in succ[n]} in[n']$ (backward)
- $in[n] := gen[n] \cup (out[n] - kill[n])$

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

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

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

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

Each analysis solves constraints over some ***domain*** of facts.

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- out[n] := $\bigcup_{n' \in succ[n]}$ in[n'] (backward)
- in[n] := gen[n] ∪ (out[n] - kill[n])

The "flow function" (i.e. effect of an instruction on the facts) can often be defined by gen and kill.

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

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

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

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

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

– out[n] := $\bigcup_{n' \in succ[n]}$ in[n'] (backward)
– in[n] := gen[n] ∪ (out[n] - kill[n])

Backward analyses define out[] in terms of in[].

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

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

Forward analyses define in[] in terms of out[].

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

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

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- $out[n] := \bigcup_{n' \in succ[n]} in[n']$ (backward)
- $in[n] := gen[n] \cup (out[n] - kill[n])$

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

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

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

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

Each domain of facts comes equipped with a way of aggregating information.

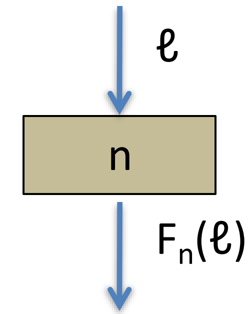
Common Features

- 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)

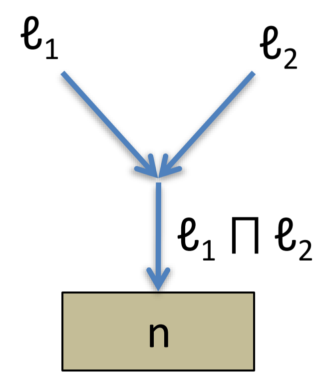
(Forward) Dataflow Analysis Framework

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

$$\text{in}[n] := \textstyle\prod_{n' \in \text{pred}[n]} \text{out}[n']$$

$$\text{out}[n] := F_n(\text{in}[n])$$

  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

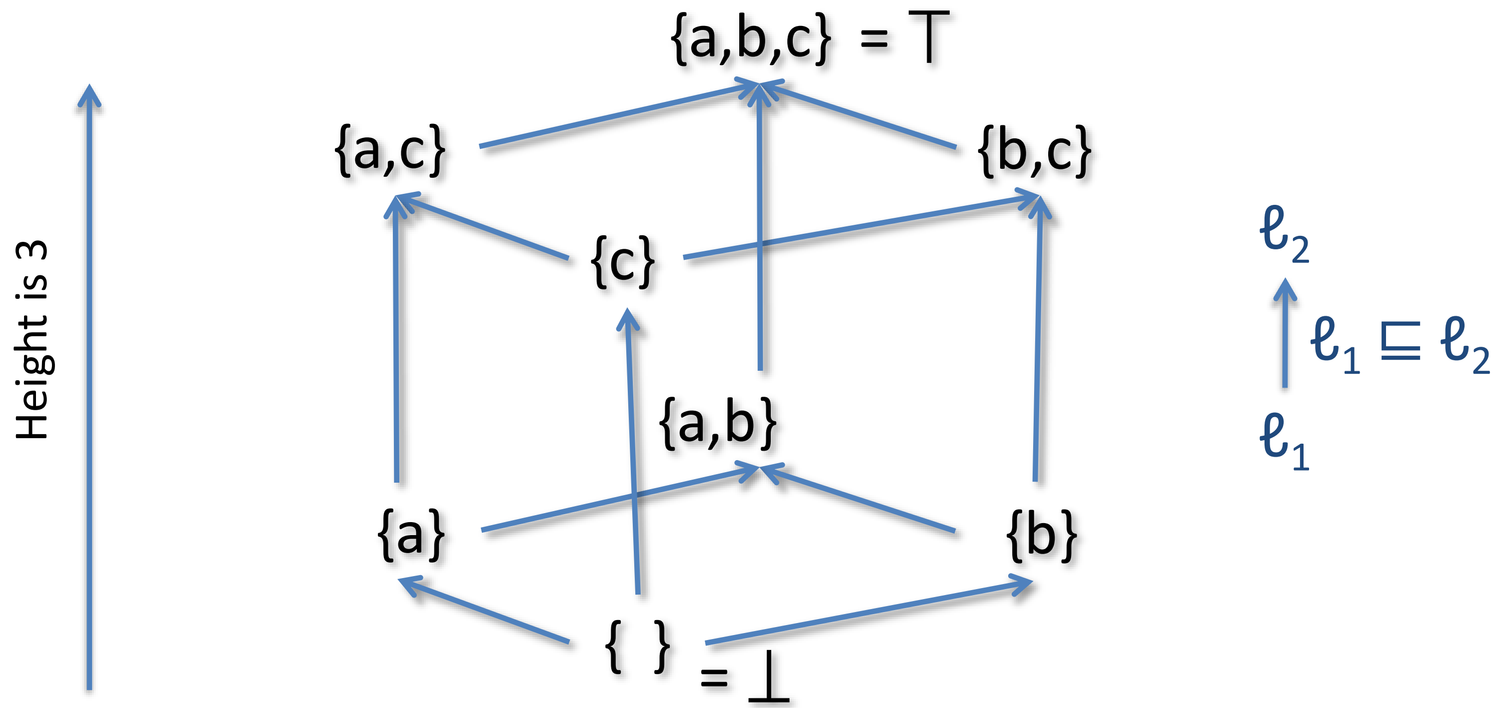
- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq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

- 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 $\sqsubseteq$ 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 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

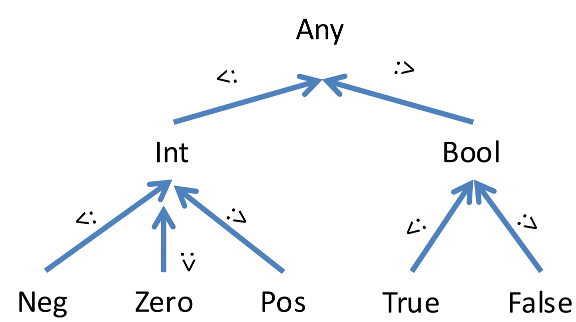
- Note that **F** is consistent: $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\ldots \sqsubseteq \mathbf{F}(\mathbf{F}(\mathbf{X}_0)) \sqsubseteq \mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$

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

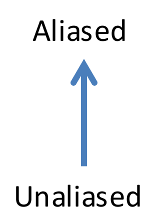
Building Lattices?

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

“Classic” Constant Propagation

- Constant propagation can be formulated as a dataflow analysis.

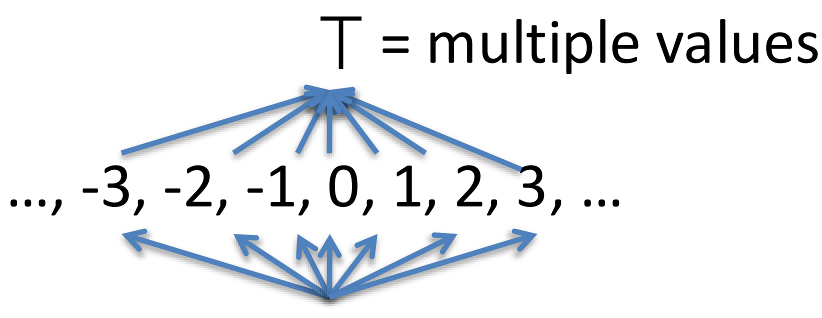
- Idea: propagate and fold integer constants in one pass:

```
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 $x = y \text{ op } z$
- $F(\ell_x, \ell_y, \ell_z) = ?$

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

"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."

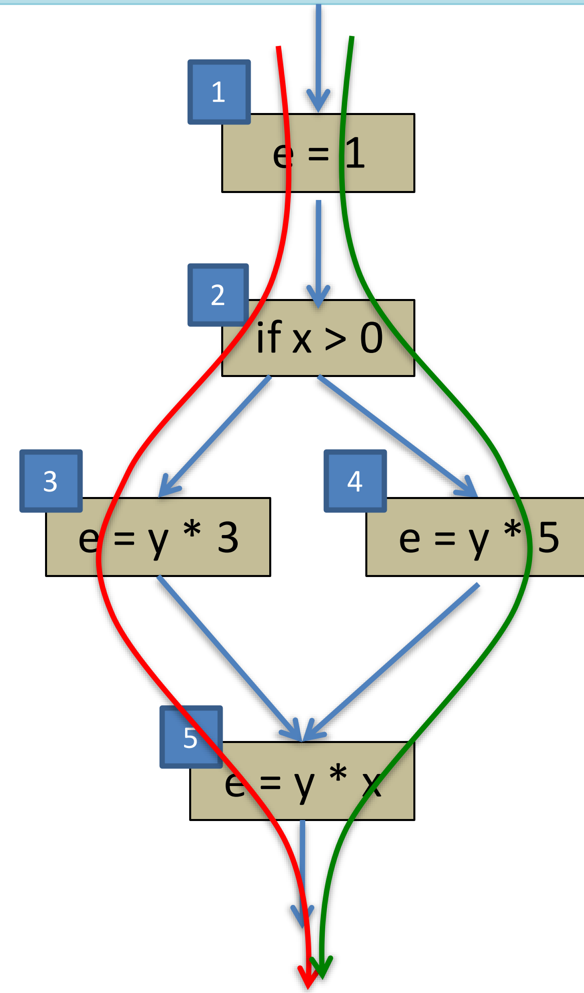
- Flow functions for the other nodes are easy…
- Monotonic?
- Distributes over meets?

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:
  $\sqcap_{p \in \text{paths\_to}[n]} \ell_p$



Best answer here is:

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

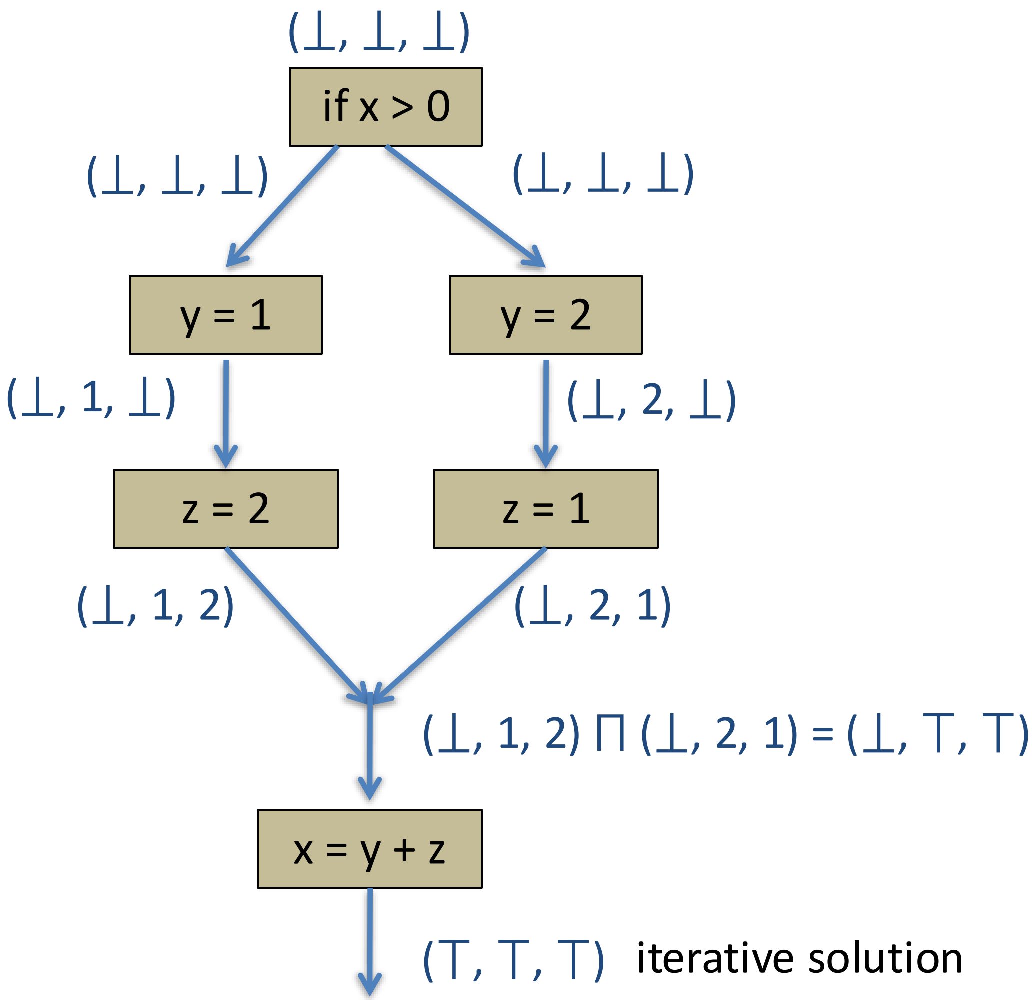
What about quality of iterative solution?

- Does the iterative solution: $out[n] = F_n(\sqcap_{n' \in pred[n]} out[n'])$ compute the MOP solution?
- MOP Solution: $\sqcap_{p \in paths\_to[n]} \ell_p$
- Answer: Yes, *if* the flow functions *distribute* over $\sqcap$
  - Distributive means: $\sqcap_i F_n(\ell_i) = F_n(\sqcap_i \ell_i)$
  - 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.

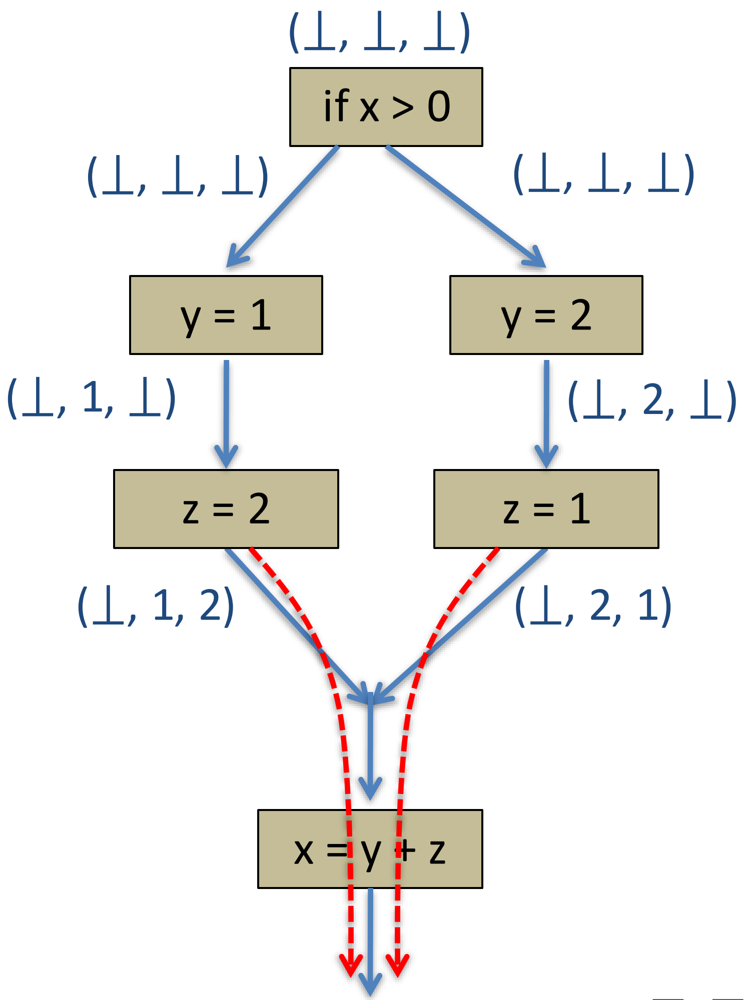
Reaching Definitions is MOP

- $F_n[x] = gen[n] \cup (x - kill[n])$
- 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

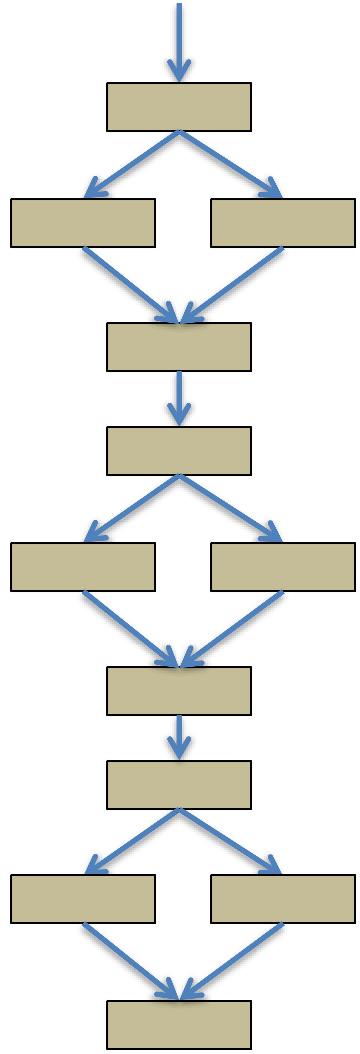


MOP solution $(3, 1, 2) \sqcap (3, 2, 1) = (3, \top, \top)$

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

REGISTER ALLOCATION

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?

Register Allocation Problem

- Given: an IR program that uses an unbounded number of temporaries
  - e.g. the uids of our LLVM programs
- Find: a mapping from temporaries to machine registers such that
  - program semantics is preserved (i.e. the behavior is the same)
  - register usage is maximized
  - moves between registers are minimized
  - calling conventions / architecture requirements are obeyed
- Stack Spilling
  - If there are k registers available and m > k temporaries are live at the same time, then not all of them will fit into registers.
  - So: "spill" the excess temporaries to the stack.

Linear-Scan Register Allocation

Simple, greedy register-allocation strategy:

1. Compute liveness information: live(x)
   – recall: live(x)is the set of uids that are live on entry to x's definition
2. Let pal be the set of usable registers
   – usually reserve a couple for spill code [our implementation uses rax,rcx]
3. Maintain "layout" uid_loc that maps uids to locations
   – locations include registers and stack slots n, starting at n=0
4. Scan through the program. For each instruction that defines a uid x
   – used = {r | reg r = uid_loc(y) s.t. y ∈ live(x)}
   – available = pal - used
   – If available is empty: *// no registers available, spill*
     uid_loc(x) := slot n ; n = n + 1
   – Otherwise, pick r in available: *// choose an available register*
     uid_loc(x) := reg r

For HW6

- HW 6 implements two naive register allocation strategies:
  - none: spill all registers
  - greedy: uses linear scan
- Also offers choice of liveness
  - trivial: assume all variables are live everywhere
  - dataflow: use the dataflow algorithms
- Your job: do "better" than these.
  - To beat "greedy" on small programs – it is necessary to take into account the calling conventions

- Quality Metric - *lower* score is better:
  - total number of memory accesses
    (Ind2 and Ind3 operands, Push/Pop)
  - ties broken by total number of instructions
- Linear scan is OK
  - but… how can we do better?