Lecture 23

CIS 341: COMPILERS

Announcements

- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Available Soon
  - Due: Wednesday, April 27^{th}

- Final Exam:
  - According to registrar: Monday, May 2^{nd} noon - 2:00pm

CODE ANALYSIS

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

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

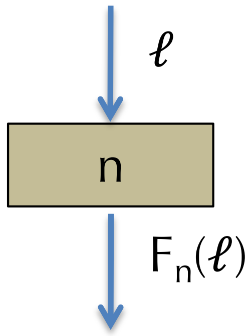
Common Features

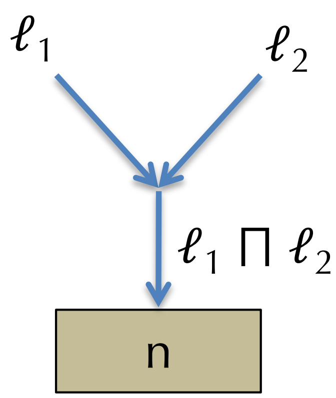
- All of these analyses have a *domain* over which they solve constraints.
  - Liveness, the domain is sets of variables
  - Reaching defns., Available exprs. the domain is sets of nodes
- Each analysis has a notion of gen[n] and kill[n]
  - Used to explain how information propagates across a node.
- 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 (∪)
  - 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 $F_n : \mathcal{L} \to \mathcal{L}$
   - So far we've seen $F_n(\ell)$ = gen[n] ∪ (ℓ - 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 $\ell_1 \sqcap \ell_2$"
   - $\text{in}[n] := \bigsqcap_{n' \in \text{pred}[n]} \text{out}[n']$





Generic Iterative (Forward) Analysis

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

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

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

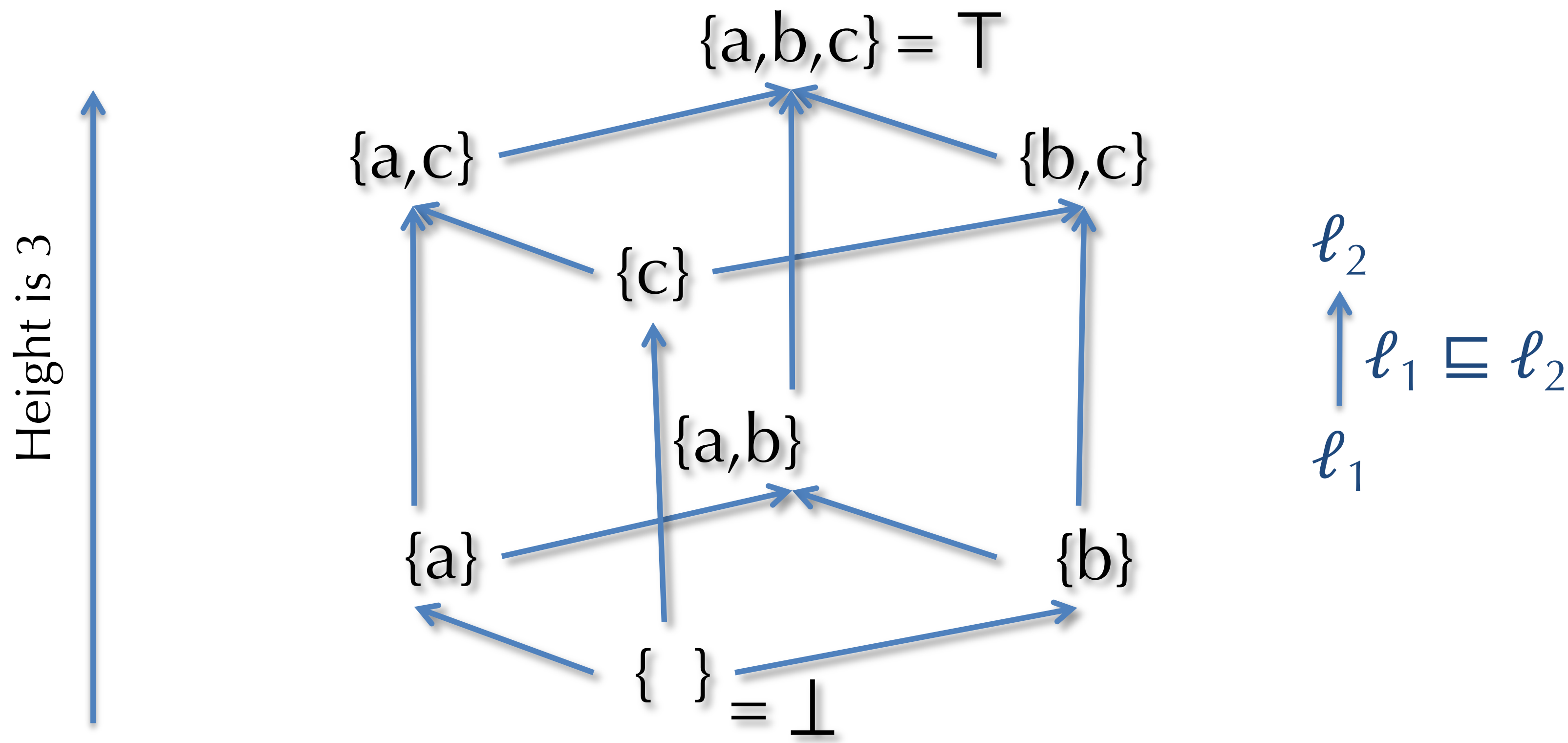
- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq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

- 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 $\sqsubseteq$ 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

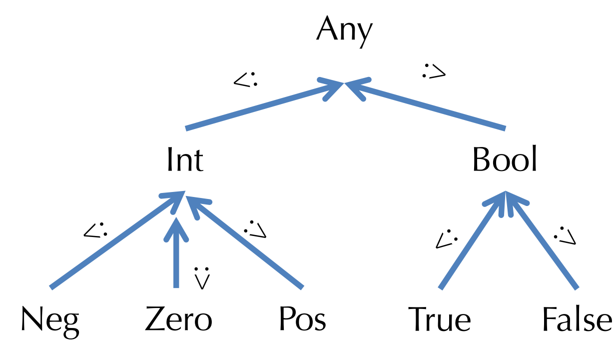
- Note that **F** is consistent: $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
  – So each iteration moves at least one step down the lattice (for some component of the vector)
  – $\ldots \sqsubseteq \mathbf{F}(\mathbf{F}(\mathbf{X}_0)) \sqsubseteq \mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$

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

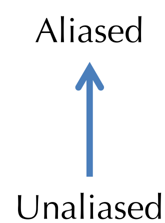
Building Lattices?

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

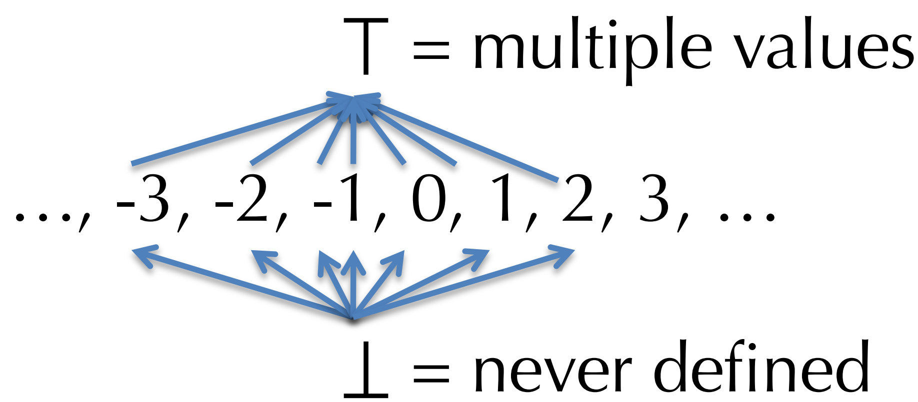
- Idea: propagate and fold integer constants in one pass:

```
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?

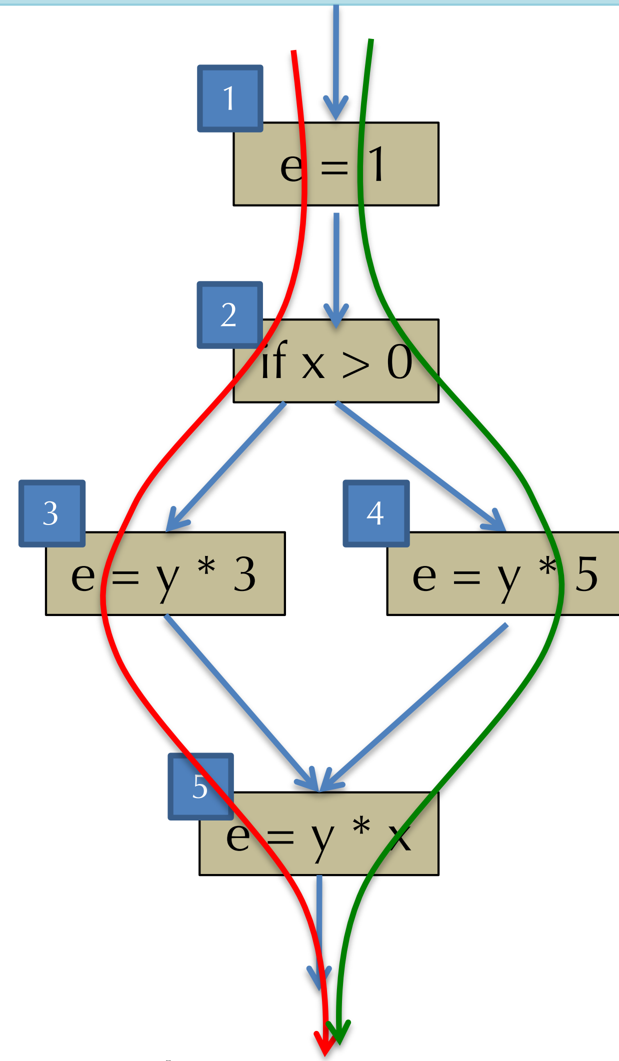
QUALITY OF DATAFLOW ANALYSIS SOLUTIONS

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: $\ell_{p1} = F_{nk}(\ldots F_{n2}(F_{n1}(F_{n0}(\top)))\ldots)$
- 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:

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

What about quality of iterative solution?

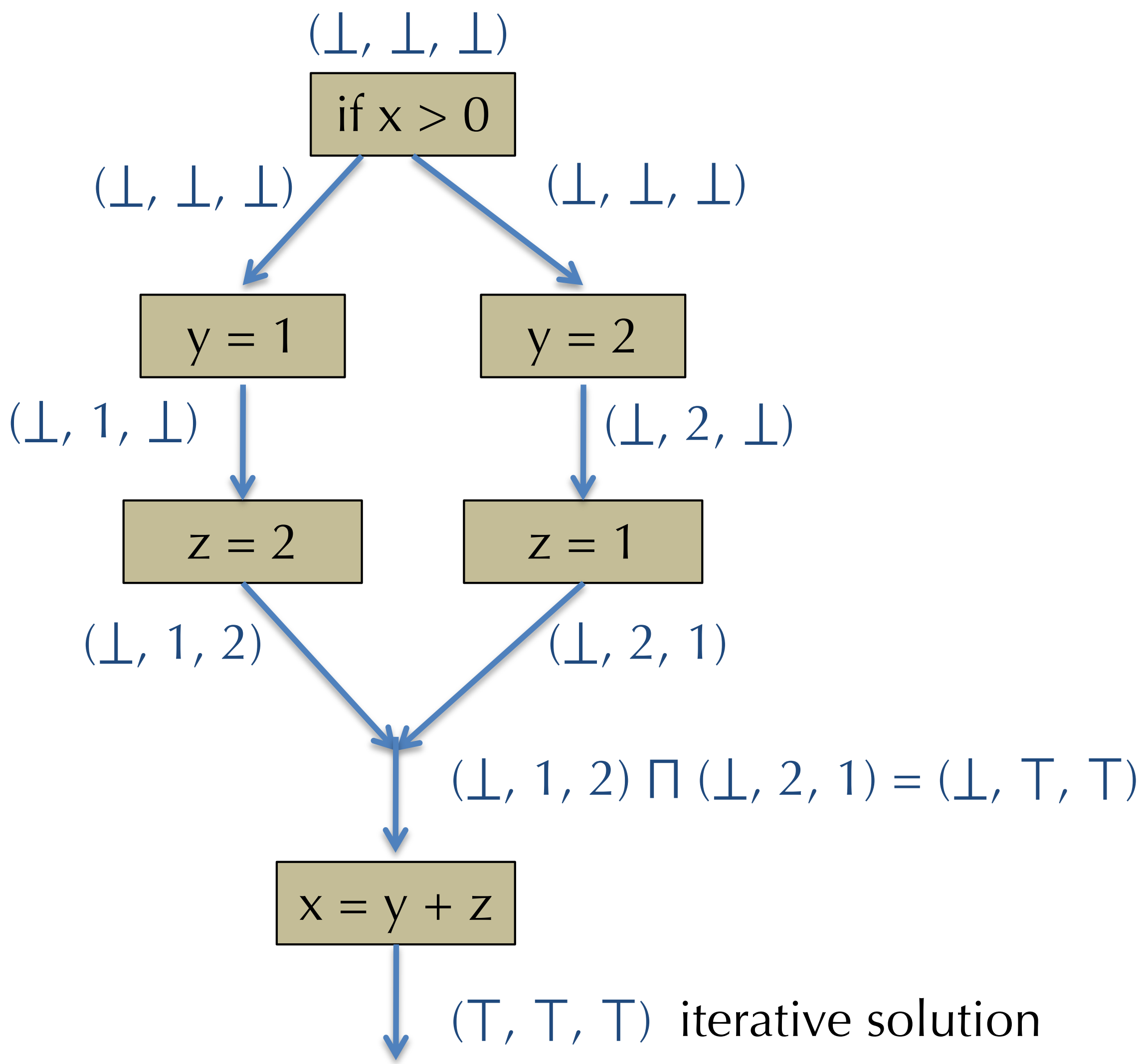
- Does the iterative solution: $out[n] = F_n(\sqcap_{n' \in pred[n]} out[n'])$ compute the MOP solution?
- MOP Solution: $\sqcap_{p \in paths\_to[n]} \ell_p$
- Answer: Yes, *if* the flow functions *distribute* over $\sqcap$
  - Distributive means: $\sqcap_i F_n(\ell_i) = F_n(\sqcap_i \ell_i)$
  - Proof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

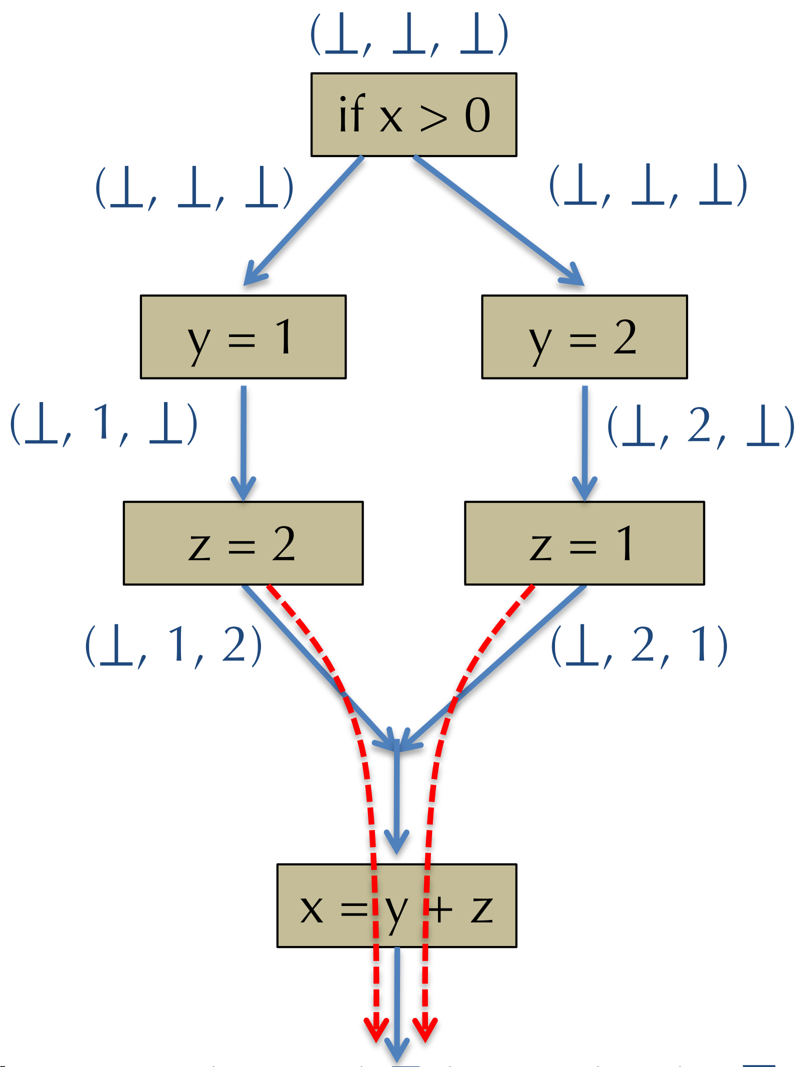
- $F_n[x] = gen[n] \cup (x - kill[n])$
- Does F_n distribute over meet $\sqcap = \cup$?
- $F_n[x \sqcap y]$
  $= gen[n] \cup ((x \cup y) - kill[n])$
  $= gen[n] \cup ((x - kill[n]) \cup (y - kill[n]))$
  $= (gen[n] \cup (x - kill[n])) \cup (gen[n] \cup (y - kill[n]))$
  $= F_n[x] \cup F_n[y]$
  $= F_n[x]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

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

Constprop Iterative Solution



MOP Solution ≠ Iterative Solution

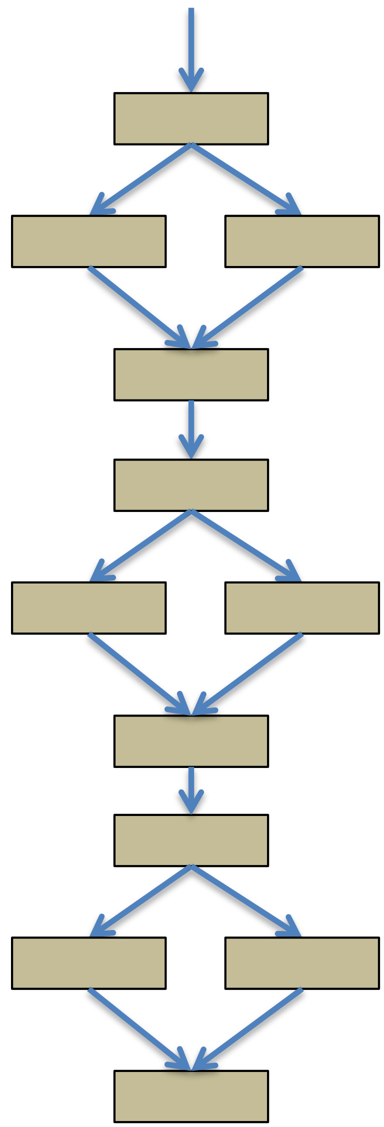


MOP solution $(3, 1, 2) \sqcap (3, 2, 1) = (3, \top, \top)$

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

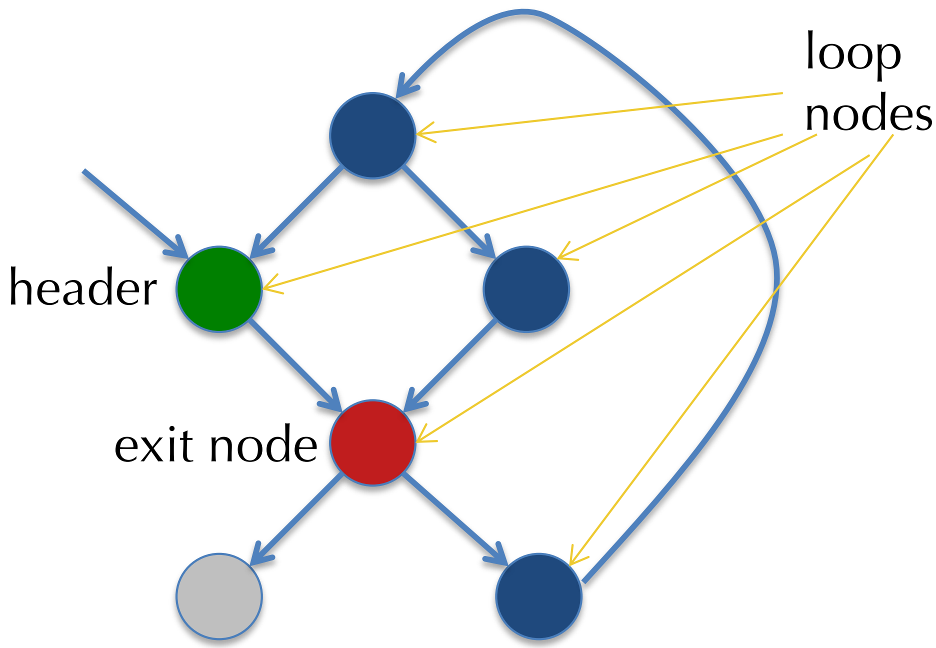
LOOPS AND DOMINATORS

Loops in Control-flow Graphs

- Taking into account loops is important for optimizations.
  - The 90/10 rule applies, so optimizing loop bodies is important
- Should we apply loop optimizations at the AST level or at a lower representation?
  - Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is good to interleave them.
- Loops may be hard to recognize at the quadruple / LLVM IR level.
  - Many kinds of loops: while, do/while, for, continue, goto…
- Problem: *How do we identify loops in the control-flow graph?*

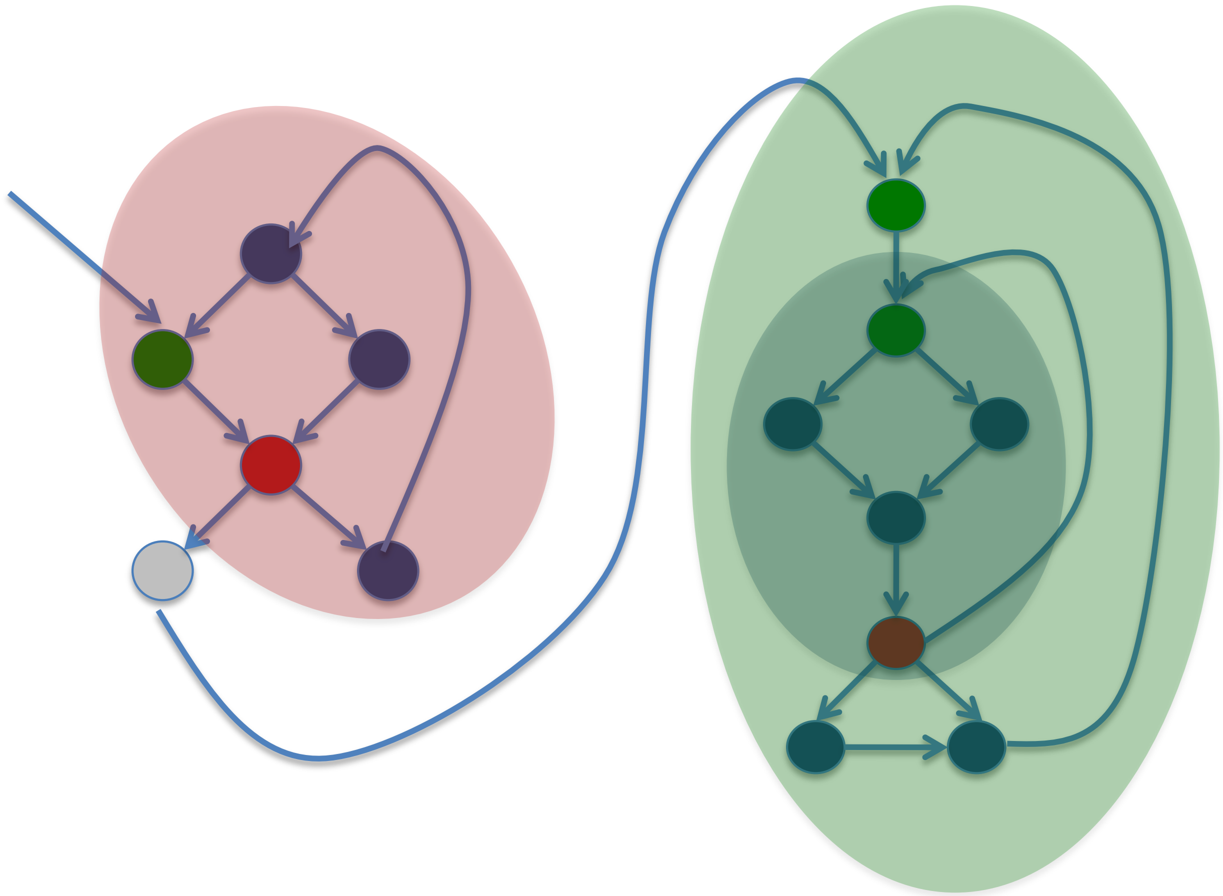
Definition of a Loop

- A *loop* is a set of nodes in the control flow graph.
  - One distinguished entry point called the *header*
- Every node is reachable from the header & the header is reachable from every node.
  - A loop is a *strongly connected component*
- No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes

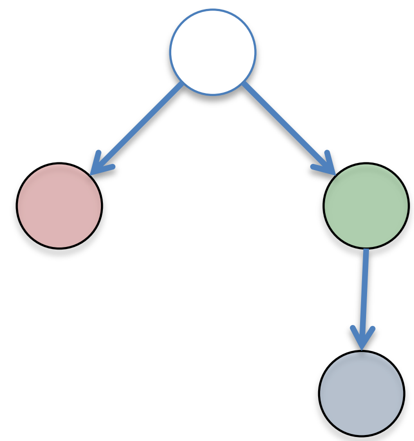


Nested Loops

- Control-flow graphs may contain many loops
- Loops may contain other loops:



Control Tree:



The control tree depicts the nesting structure of the program.

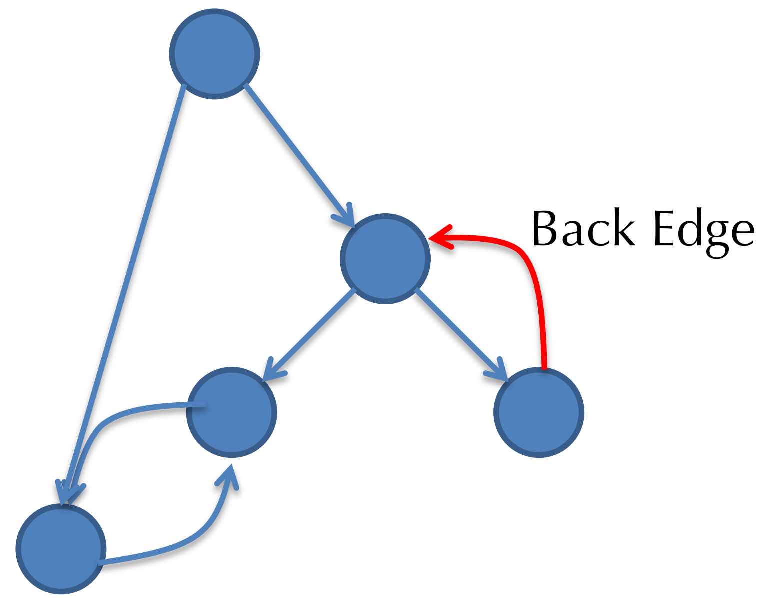
Control-flow Analysis

- Goal: Identify the loops and nesting structure of the CFG.

- Control flow analysis is based on the idea of *dominators*:
- Node A *dominates* node B if the only way to reach B from the start node is through node A.

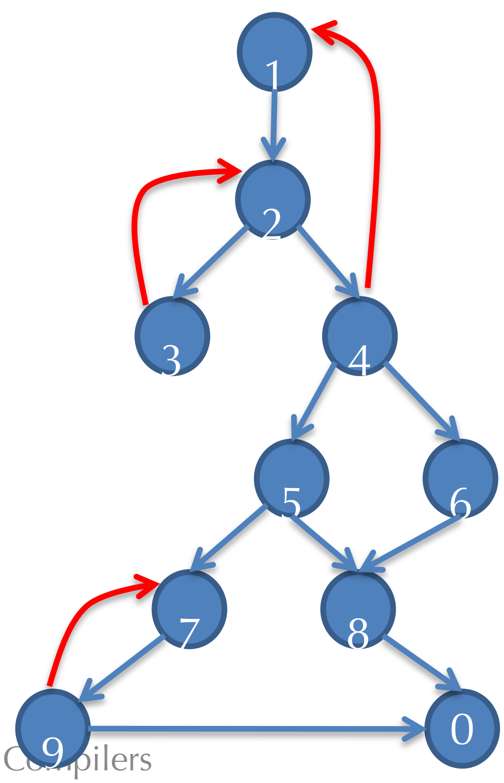
- An edge in the graph is a *back edge* if the target node dominates the source node.

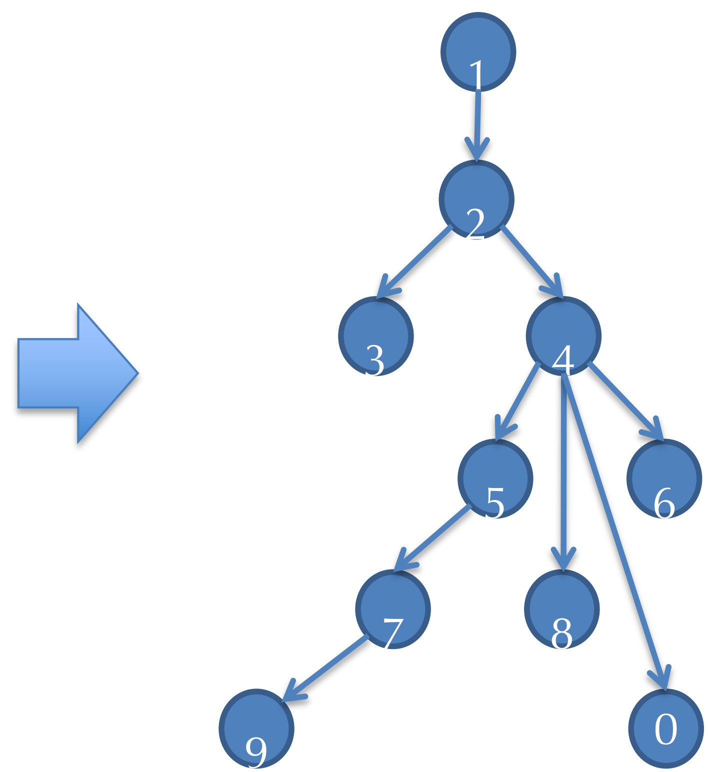
- A loop contains at least one back edge.



Dominator Trees

- Domination is transitive:
  - if A dominates B and B dominates C then A dominates C
- Domination is anti-symmetric:
  - if A dominates B and B dominates A then A = B
- Every flow graph has a dominator tree
  - The Hasse diagram of the dominates relation





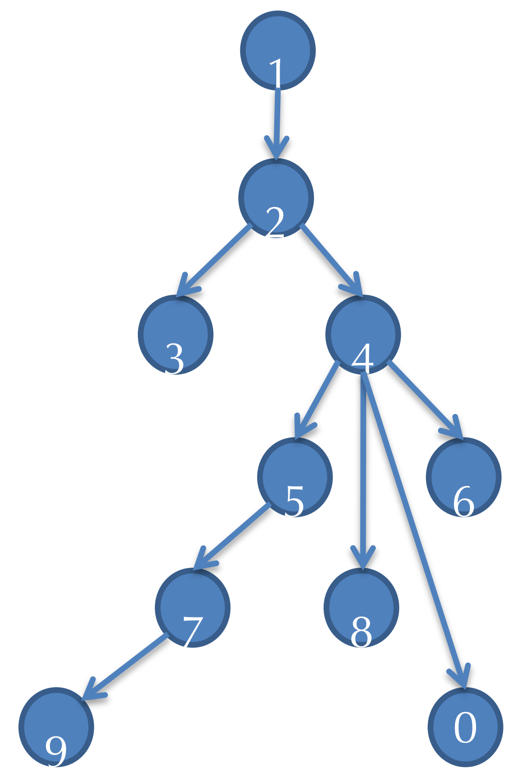
Dominator Dataflow Analysis

- We can define Dom[n] as a forward dataflow analysis.
  - Using the framework we saw earlier: Dom[n] = out[n] where:
- "A node B is dominated by another node A if A dominates *all* of the predecessors of B."
  - $\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']$
- "Every node dominates itself."
  - $\text{out}[n] := \text{in}[n] \cup \{n\}$
- Formally: $\mathcal{L}$ = set of nodes ordered by $\subseteq$
  - $\top = \{\text{all nodes}\}$
  - $F_n(x) = x \cup \{n\}$
  - $\sqcap$ is $\cap$
- Easy to show monotonicity and that F_n distributes over meet.
  - So algorithm terminates and is MOP

Improving the Algorithm

- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
  - e.g. Dom[8] = {1,2,4,8}, Dom[7] = {1,2,4,5,7}
  - There is a lot of sharing among the nodes

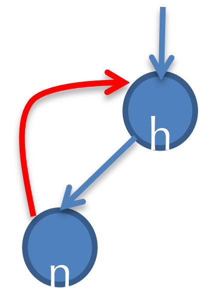
- More efficient way to represent Dom sets is to store the dominator *tree*.
  - doms[b] = immediate dominator of b
  - doms[8] = 4, doms[7] = 5
- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
  - Traverse up tree, looking for least common ancestor:
  - Dom[8] ∩Dom[7] = Dom[4]



- See: "A Simple, Fast Dominance Algorithm" Cooper, Harvey, and Kennedy

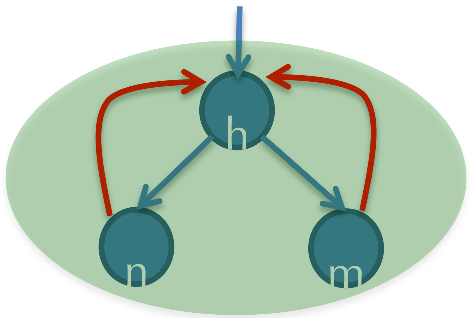
Completing Control-flow Analysis

- Dominator analysis identifies *back edges*:
  - Edge n → h where h dominates n
- Each back edge has a *natural loop*:
  - h is the header
  - All nodes reachable from h that also reach n without going through h



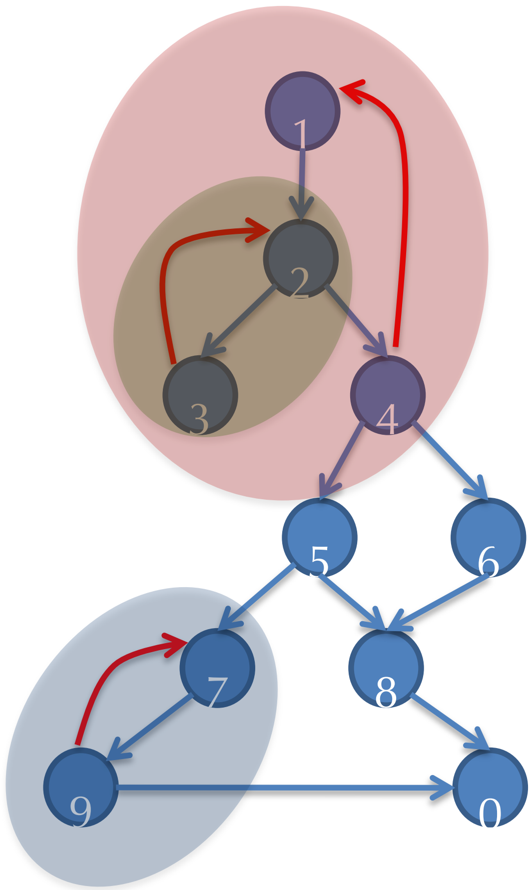
- For each back edge n → h, find the natural loop:
  - {n′ | n is reachable from n′ in G – {h}} ∪ {h}

- Two loops may share the same header: merge them



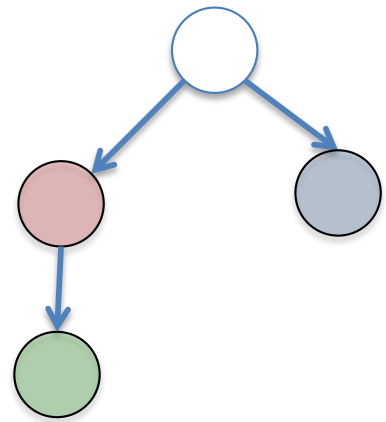
- Nesting structure of loops is determined by set inclusion
  - Can be used to build the control tree

Example Natural Loops



Natural Loops

Control Tree:



The control tree depicts the nesting structure of the program.

Uses of Control-flow Information

- Loop nesting depth plays an important role in optimization heuristics.
  - Deeply nested loops pay off the most for optimization.
- Need to know loop headers / back edges for doing
  - loop invariant code motion
  - loop unrolling
- Dominance information also plays a role in converting to SSA form
  - Used internally by LLVM to do register allocation.

Phi nodes

Alloc "promotion"

Register allocation

REVISITING SSA

Single Static Assignment (SSA)

- LLVM IR names (via `%uids`) *all* intermediate values computed by the program.
- It makes the order of evaluation explicit.
- Each `%uid` is assigned to only once
  - Contrast with the mutable quadruple form
  - Note that dataflow analyses had these kill[n] sets because of updates to variables…
- Naïve implementation of backend: map `%uids` to stack slots
- Better implementation: map `%uids` to registers (as much as possible)

- Question: How do we convert a source program to make maximal use of `%uids`, rather than alloca-created storage?
  - two problems: control flow & location in memory

- Then: How do we convert SSA code to x86, mapping `%uids` to registers?
  - Register allocation.

Alloca vs. %UID

- Current compilation strategy:

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
%x = alloca i64
%y = alloca i64
store i64* %x, 3
store i64* %y, 0
%x1 = load %i64* %x
%tmp1 = add i64 %x1, 1
store i64* %x, %tmp1
%x2 = load %i64* %x
%tmp2 = add i64 %x2, 2
store i64* %y, %tmp2
```

- Directly map source variables into `%uids`?

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

```
%x1 = add i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %x2, 2
```

- Does this always work?

What about If-then-else?

- How do we translate this into SSA?

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %z2 = %add i64 ???, 3
```

- What do we put for `???`

Phi Functions

- Solution: ϕ functions
  - Fictitious operator, used only for analysis
    - implemented by Mov at x86 level
  - Chooses among different versions of a variable based on the path by which control enters the phi node.

    `%uid = phi <ty>` v_1, `<label`$_1$`>`, … , v_n, `<label`$_n$`>`

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %x4 = phi i64 %x2, %then, %x3, %else
  %z2 = %add i64 %x4, 3
```

Phi Nodes and Loops

- Importantly, the `%uids` on the right-hand side of a phi node can be defined "later" in the control-flow graph.
  - Means that `%uids` can hold values "around a loop"
  - Scope of `%uids` is defined by *dominance*

```
entry:
  %y1 = …
  %x1 = …
  br label %body

body:
  %x2 = phi i64 %x1, %entry, %x3, %body
  %x3 = add i64 %x2, %y1
  %p = icmp slt i64, %x3, 10
  br i1 %p, label %body, label %after

after:
  …
```

Alloca Promotion

- Not all source variables can be allocated to registers
  - If the address of the variable is taken (as permitted in C, for example)
  - If the address of the variable "escapes" (by being passed to a function)
- An alloca instruction is called promotable if neither of the two conditions above holds

```
entry:
  %x = alloca i64           // %x cannot be promoted
  %y = call malloc(i64 8)
  %ptr = bitcast i8* %y to i64**
  store i65** %ptr, %x      // store the pointer into the heap
```

```
entry:
  %x = alloca i64         // %x cannot be promoted
  %y = call foo(i64* %x) // foo may store the pointer into the heap
```

- Happily, most local variables declared in source programs are promotable
  - That means they can be register allocated

Converting to SSA: Overview

- Start with the ordinary control flow graph that uses allocas
  - Identify "promotable" allocas
- Compute dominator tree information
- Calculate def/use information for each such allocated variable
- Insert ϕ functions for each variable at necessary "join points"

- Replace loads/stores to alloc'ed variables with freshly-generated `%uids`

- Eliminate the now unneeded load/store/alloca instructions.

Where to Place ϕ functions?

- Need to calculate the "Dominance Frontier"

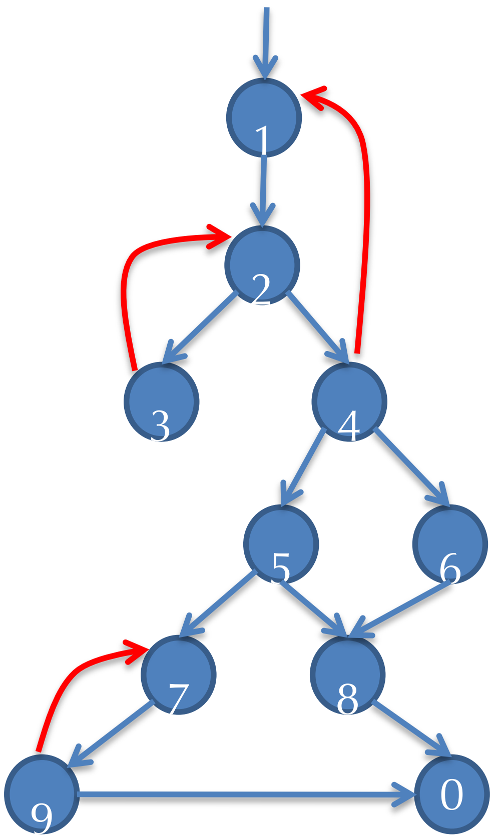
- Node A ***strictly dominates*** node B if A dominates B and A ≠ B.
  - Note: A does not strictly dominate B if A does not dominate B or A = B.

- The ***dominance frontier*** of a node B is the set of all CFG nodes y such that B dominates a predecessor of y but does not strictly dominate y
  - Intuitively: starting at B, there is a path to y, but there is another route to y that does not go through B

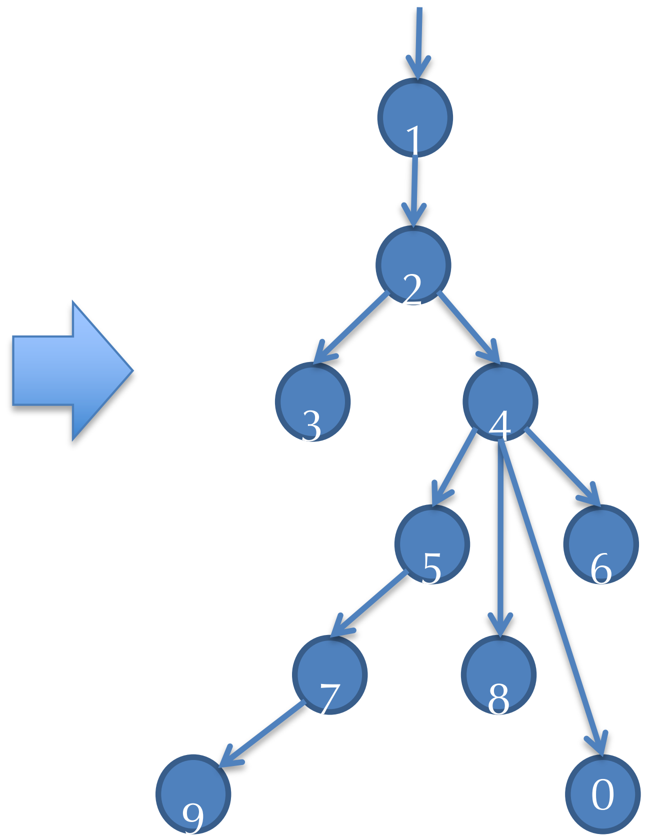
- Write DF[n] for the dominance frontier of node n.

Dominance Frontiers

- Example of a dominance frontier calculation results
- DF[1] = {1}, DF[2] = {1,2}, DF[3] = {2}, DF[4] = {1}, DF[5] = {8,0}, DF[6] = {8}, DF[7] = {7,0}, DF[8] = {0}, DF[9] = {7,0}, DF[0] = {}



Control-flow Graph



Dominator Tree

Algorithm For Computing DF[n]

- Assume that doms[n] stores the dominator tree (so that doms[n] is the *immediate dominator* of n in the tree)

- Adds each B to the DF sets to which it belongs

```
for all nodes B
    if #(pred[B]) ≥ 2                    // (just an optimization)
     for each p ∈pred[B] {
          runner := p                    // start at the predecessor of B
          while (runner ≠ doms[B])       // walk up the tree adding B
               DF[runner] := DF[runner] ∪ {B}
               runner := doms[runner]
    }
```

Insert ϕ at Join Points

- Lift the DF[n] to a set of nodes N in the obvious way:

$$DF[N] = \bigcup_{n \in N} DF[n]$$

- Suppose that at variable x is defined at a set of nodes N.

  $DF_0[N] = DF[N]$
  $DF_{i+1}[N] = DF[DF_i[N] \cup N]$

  Let J[N] be the *least fixed point* of the sequence:

$$DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots$$

  That is, $J[N] = DF_k[N]$ for some k such that $DF_k[N] = DF_{k+1}[N]$
  - J[N] is called the "join points" for the set N
- We insert ϕ functions for the variable x at each node in J[N].
  - $x = \phi(x, x, \ldots, x);$ (one "x" argument for each predecessor of the node)
  - In practice, J[N] is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes, just as in the dataflow solver.

- Intuition:
  - If N is the set of places where x is modified, then DF[N] is the places where phi nodes need to be added, but those also "count" as modifications of x, so we need to insert the phi nodes to capture those modifications too…

Example Join-point Calculation

- Suppose the variable x is modified at nodes 3 and 6
  - Where would we need to add phi nodes?

- $DF_0[\{3,6\}] = DF[\{3,6\}] = DF[3] \cup DF[6] = \{2,8\}$
- $DF_1[\{3,6\}]$

$$\begin{aligned} &= DF[DF_0\{3,6\} \cup \{3,6\}] \\ &= DF[\{2,3,6,8\}] \\ &= DF[2] \cup DF[3] \cup DF[6] \cup DF[8] \\ &= \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\} \end{aligned}$$

- $DF_2[\{3,6\}]$

$$\begin{aligned} &= \ldots \\ &= \{1,2,8,0\} \end{aligned}$$

- So $J[\{3,6\}] = \{1,2,8,0\}$ and we need to add phi nodes at those four spots.

Phi Placement Alternative

- Less efficient, but easier to understand:

- Place phi nodes "maximally" (i.e. at every node with > 2 predecessors)

- If all values flowing into phi node are the same, then eliminate it:

```
%x = phi   t %y, %pred1   t %y  %pred2  … t %y %predK
// code that uses %x
⇒
// code with %x replaced by %y
```

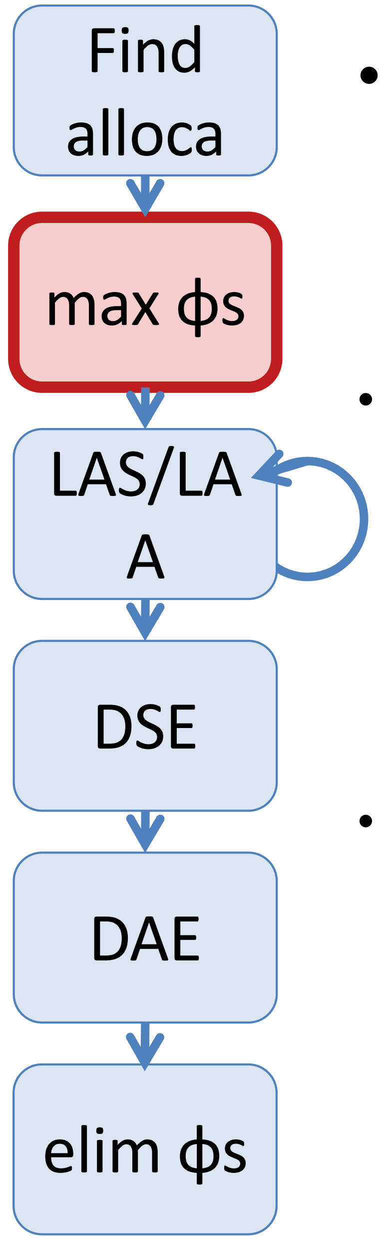
- Interleave with other optimizations
  - copy propagation
  - constant propagation
  - etc.

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2:


    store 1, %p

    br %l3
```

```
l3:


    %x = load %p
    ret %x
```



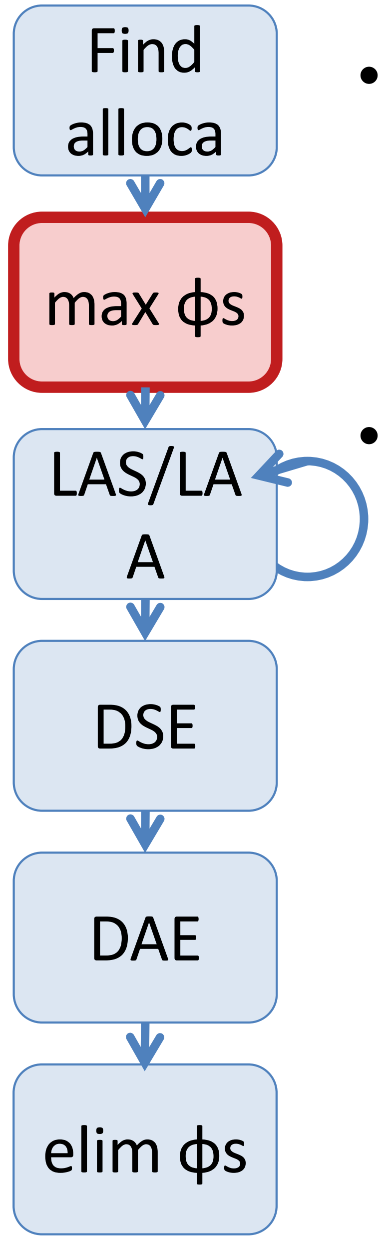
- How to place phi nodes without breaking SSA?
- Note: the "real" implementation combines many of these steps into one pass.
  - Places phis directly at the dominance frontier
- This example also illustrates other common optimizations:
  - Load after store/alloca
  - Dead store/alloca elimination

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2:

    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3:

    %x = load %p
    ret %x
```



- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

Example SSA Optimizations

```
l₁: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x₁ = load %p
    br %b, %l₂, %l₃
```

```
l₂: %x₃ = ϕ[%x₁,%l₁]

    store 1, %p
    %x₂ = load %p
    br %l₃
```

```
l₃: %x₄ = φ[%x₁;%l₁, %x₂:%l₂]

    %x = load %p
    ret %x
```

Find alloca → **max ϕs** → LAS/LAA → DSE → DAE → elim ϕs

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block

Example SSA Optimizations

```
l_1: %p = alloca i64
     store 0, %p
     %b = %y > 0
     %x_1 = load %p
     br %b, %l_2, %l_3
```

```
l_2: %x_3 = ϕ[%x_1,%l_1]
     store %x_3, %p
     store 1, %p
     %x_2 = load %p
     br %l_3
```

```
l_3: %x_4 = φ[%x_1;%l_1, %x_2:%l_2]
     store %x_4, %p
     %x = load %p
     ret %x
```

Find alloca → **max ϕs** → LAS/LAA → DSE → DAE → elim ϕs

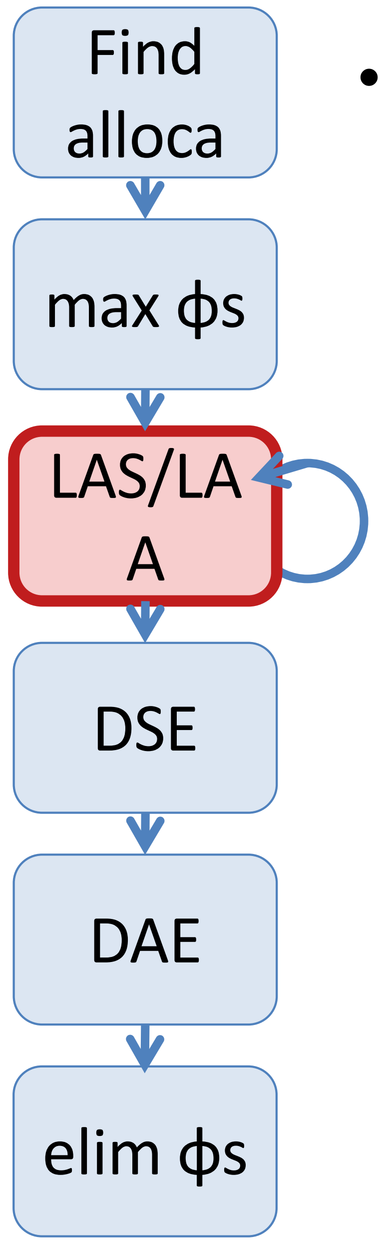
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block
  - Insert stores after φ-nodes

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[%x1,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = φ[%x1;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



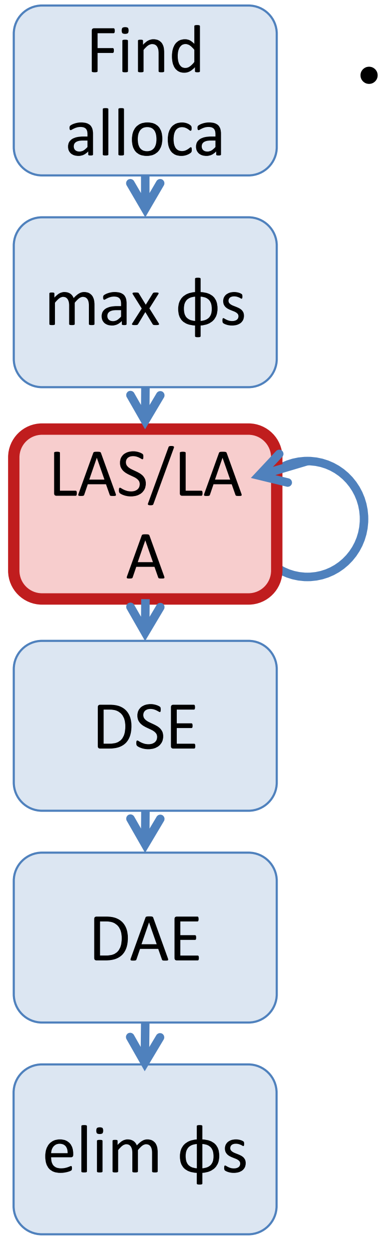
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[%x1,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

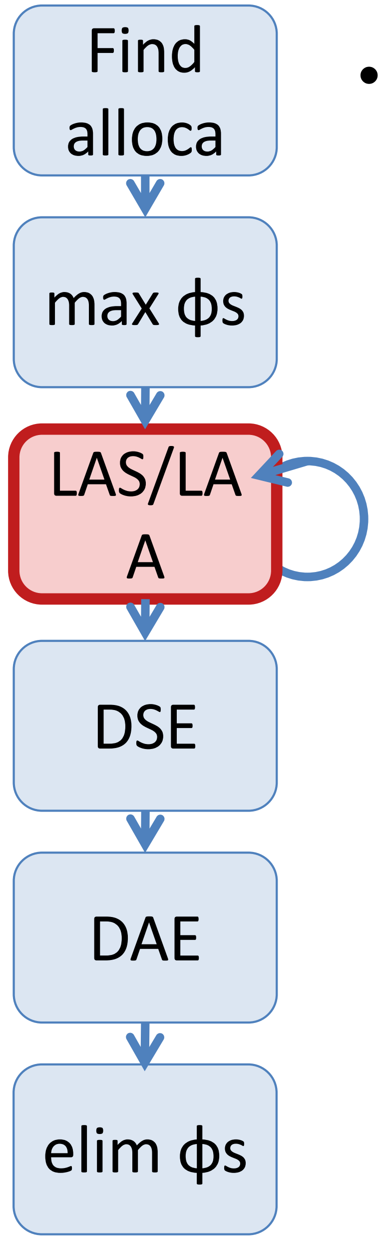
```
l3: %x4 = φ[%x1;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3

l2: %x3 = φ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[0;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



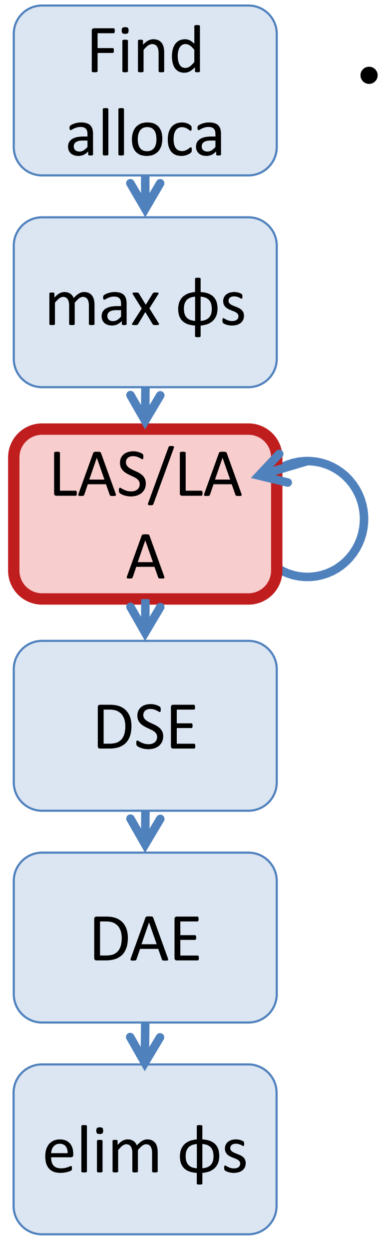
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = φ[0;%l1, %x2, %l2]
    store %x4, %p
    %x = load %p
    ret %x
```



- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

`l_1: %p = alloca i64`
`store 0, %p`
`%b = %y > 0`
`br %b, %l_2, %l_3`

`l_2: %x_3 = ϕ[0,%l_1]`
`store %x_3, %p`
$\boxed{\texttt{store 1, \%p}}$
~~`%x_2 = load %p`~~
`br %l_3`

`l_3: %x_4 = φ[0;%l_1, 1:%l_2]`
`store %x_4, %p`
`%x = load %p`
`ret %x`

Find alloca → max ϕs → LAS/LAA → DSE → DAE → elim ϕs

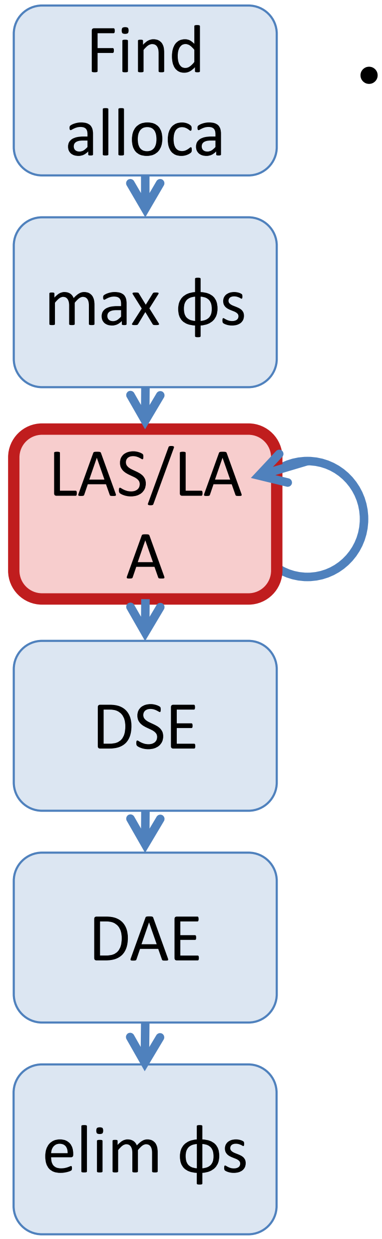
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



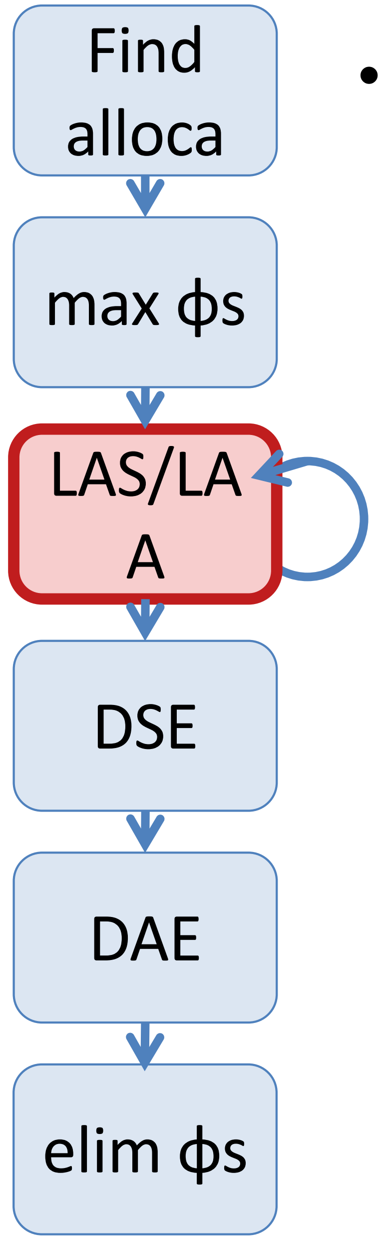
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0:%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x4
```



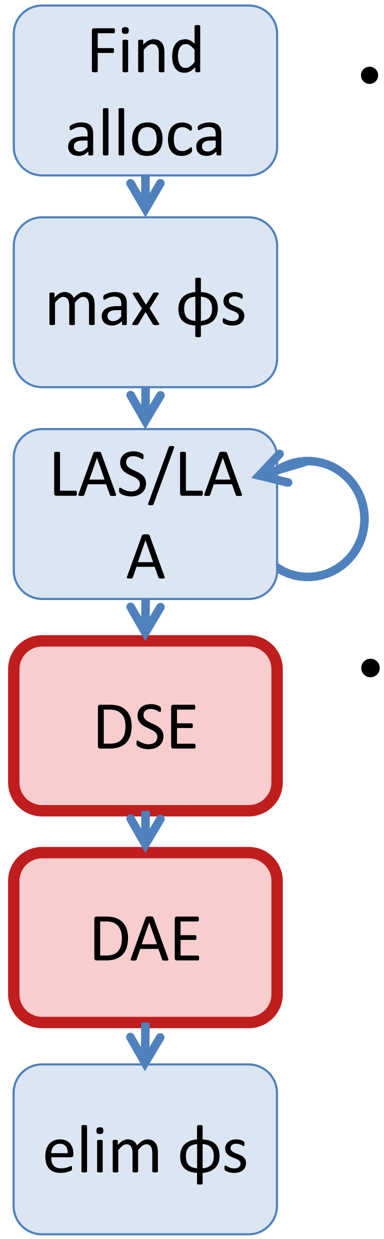
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

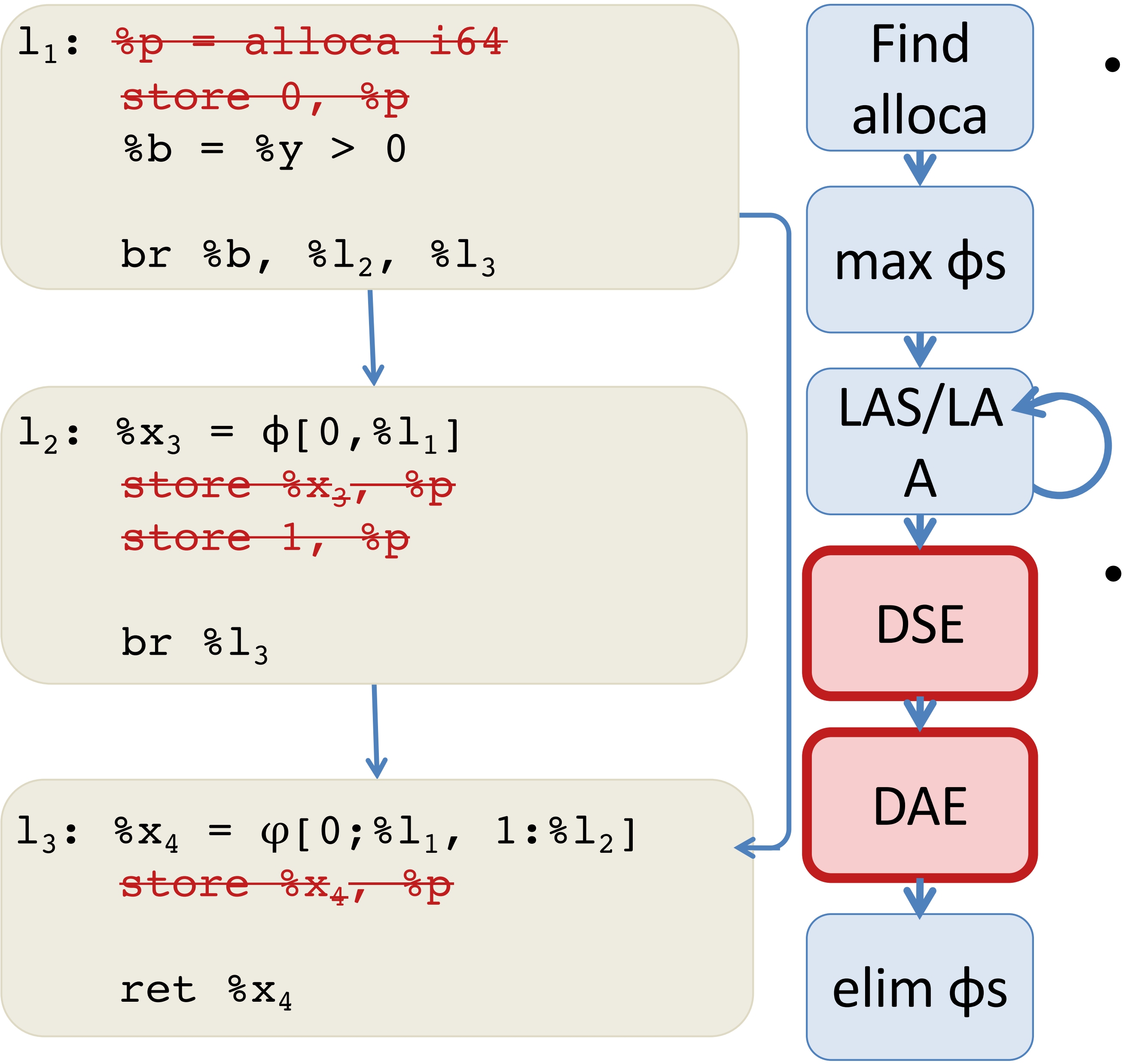
```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p

    ret %x4
```



- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

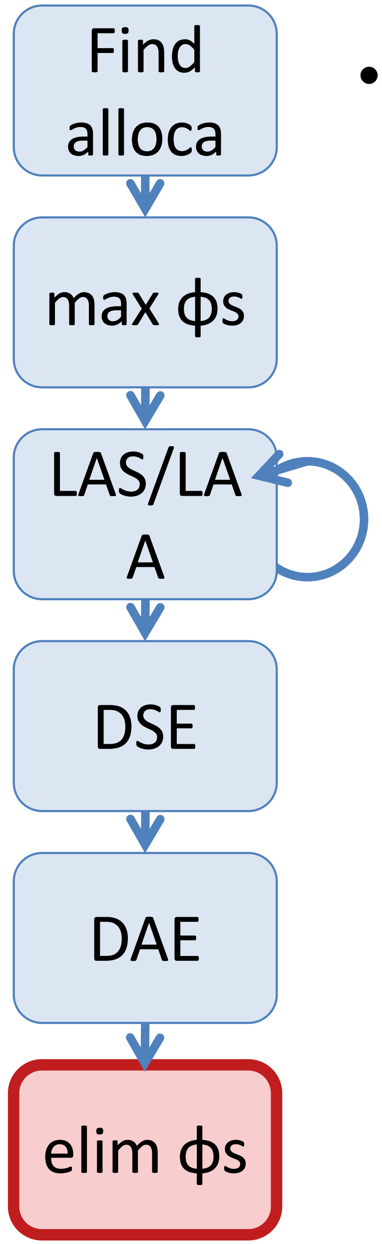
Example SSA Optimizations



- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

Example SSA Optimizations

```
l₁:
    %b = %y > 0
    br %b, %l₂, %l₃

l₂: %x₃ = ϕ[0,%l₁]

    br %l₃

l₃: %x₄ = φ[0;%l₁, 1:%l₂]

    ret %x₄
```



- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002

Example SSA Optimizations

l_1:

%b = %y > 0

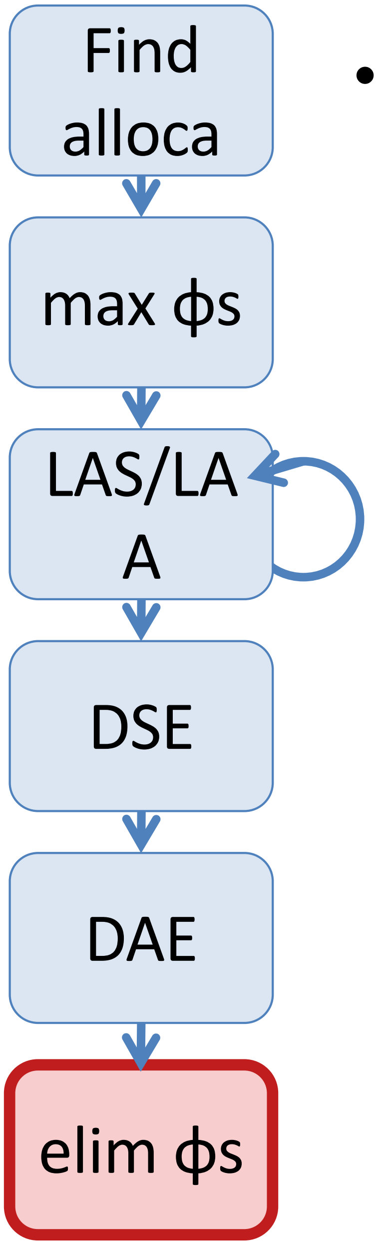
br %b, %l_2, %l_3

l_2: ~~%x_3 = ϕ[0,%l_1]~~

br %l_3

l_3: %x_4 = φ[0;%l_1, 1:%l_2]

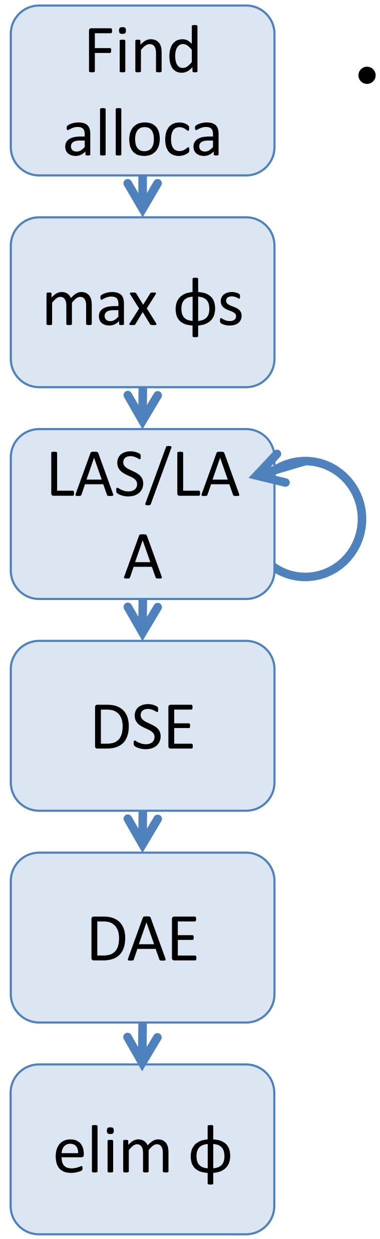
ret %x_4



- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor

Example SSA Optimizations

```
l1:
    %b = %y > 0
    br %b, %l2, %l3

l2:

    br %l3

l3: %x4 = φ[0;%l1, 1:%l2]

    ret %x4
```



- Done!

LLVM Phi Placement

- This transformation is also sometimes called register promotion
  - older versions of LLVM called this "mem2reg" memory to register promotion
- In practice, LLVM combines this transformation with *scalar replacement of aggregates* (SROA)
  - i.e. transforming loads/stores of structured data into loads/stores on register-sized data
- These algorithms are (one reason) why LLVM IR allows annotation of predecessor information in the .ll files
  - Simplifies computing the DF