Lecture 22

CIS 341: COMPILERS

Announcements

- HW 6: Dataflow Analysis and Optimizations
  - Available later today(?)
  - Due Next Thursday, April 16

- HW 7: Optimization & Experiments
  - Available next week
  - Due: April 29th

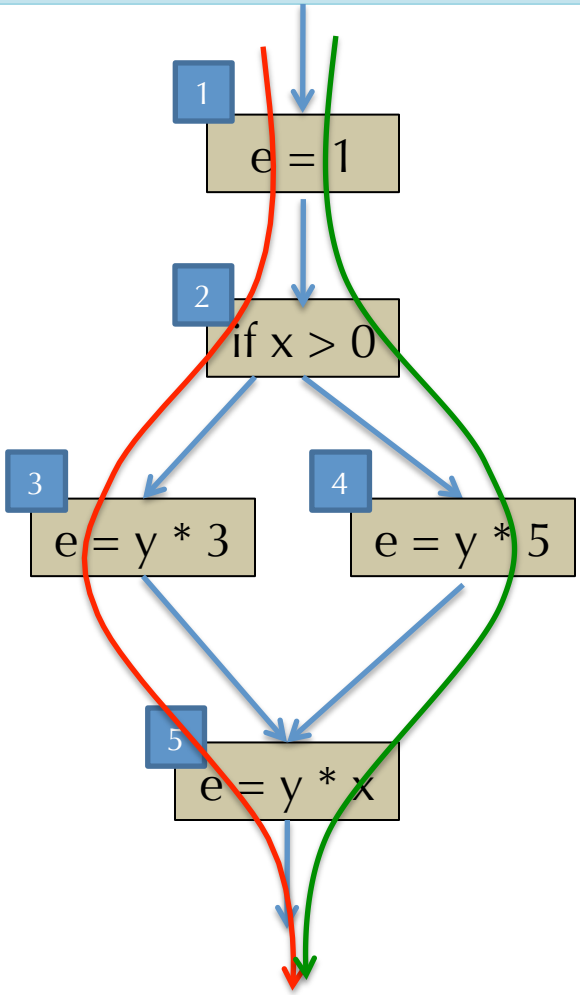
QUALITY OF DATAFLOW ANALYSIS SOLUTIONS

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: $out[n] = F_n(\sqcap_{n' \in pred[n]} out[n'])$ 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.

"Classic" Constant Propagation

- Constant propagation can be formulated as a dataflow analysis.

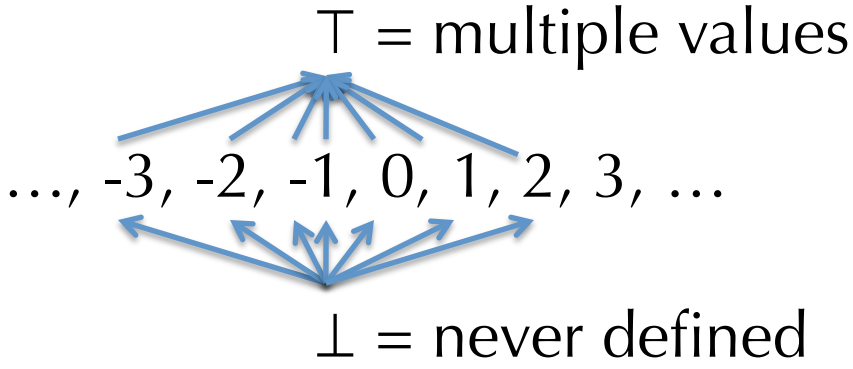
- Idea: propagate and fold integer constants in one pass:

```
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."

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

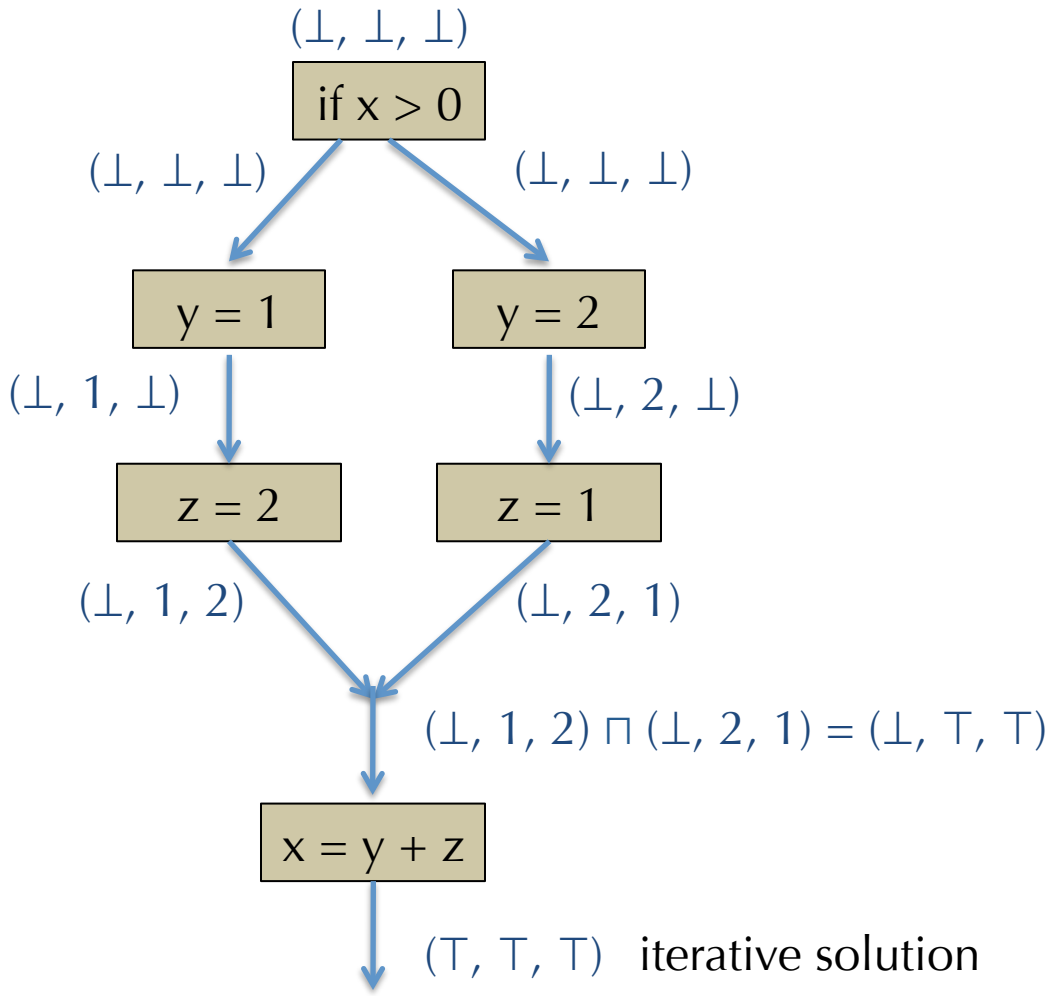
"If either input is undefined the result of the operation is too."

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

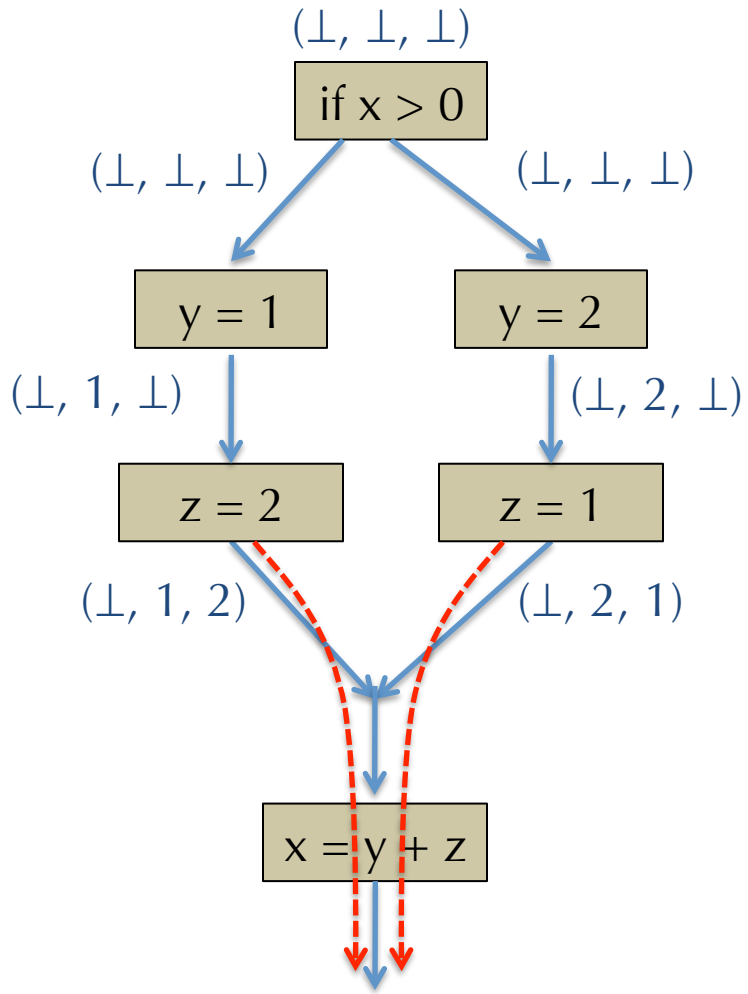
"If the inputs are known constants, calculate the output statically."

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

Iterative Solution



MOP Solution ≠ Iterative Solution

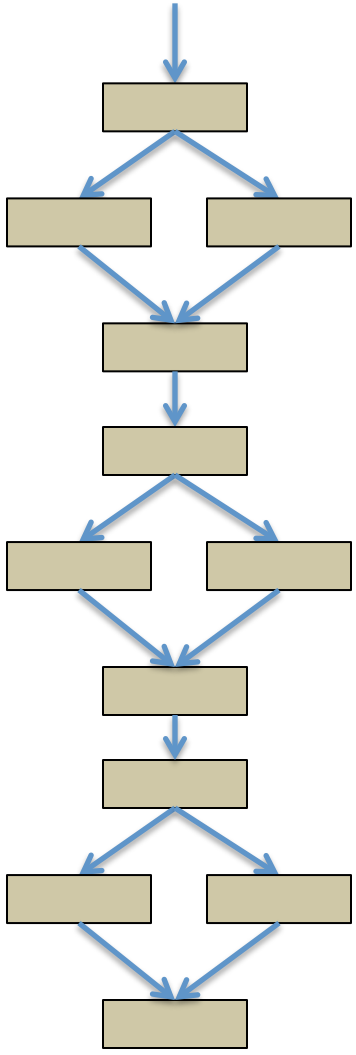


MOP solution $(3, 1, 2) \sqcap (3, 2, 1) = (3, \top, \top)$

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

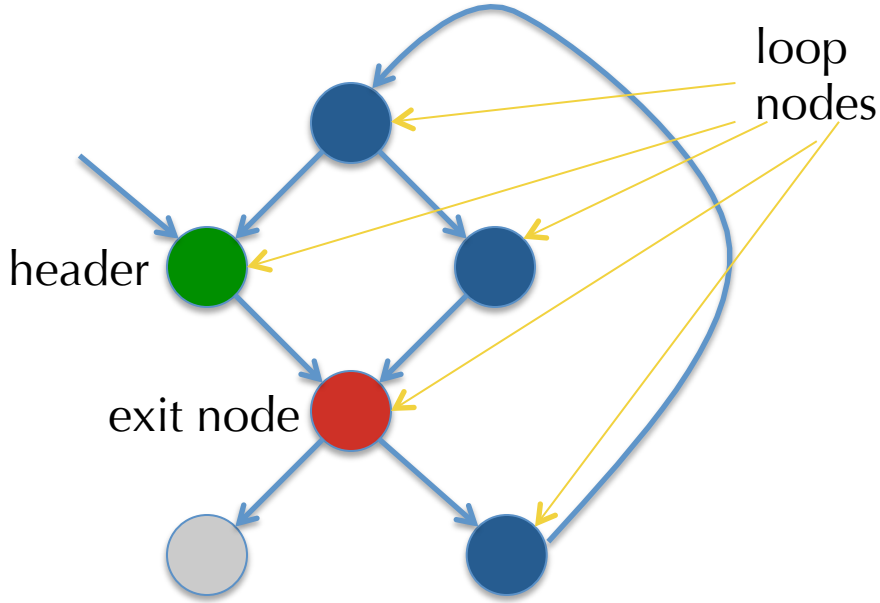
- Taking into account loops is important for optimizations.
  - The 90/10 rule applies, so optimizing loop bodies is important
- Should we apply loop optimizations at the AST level or at a lower representation?
  - Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is 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*

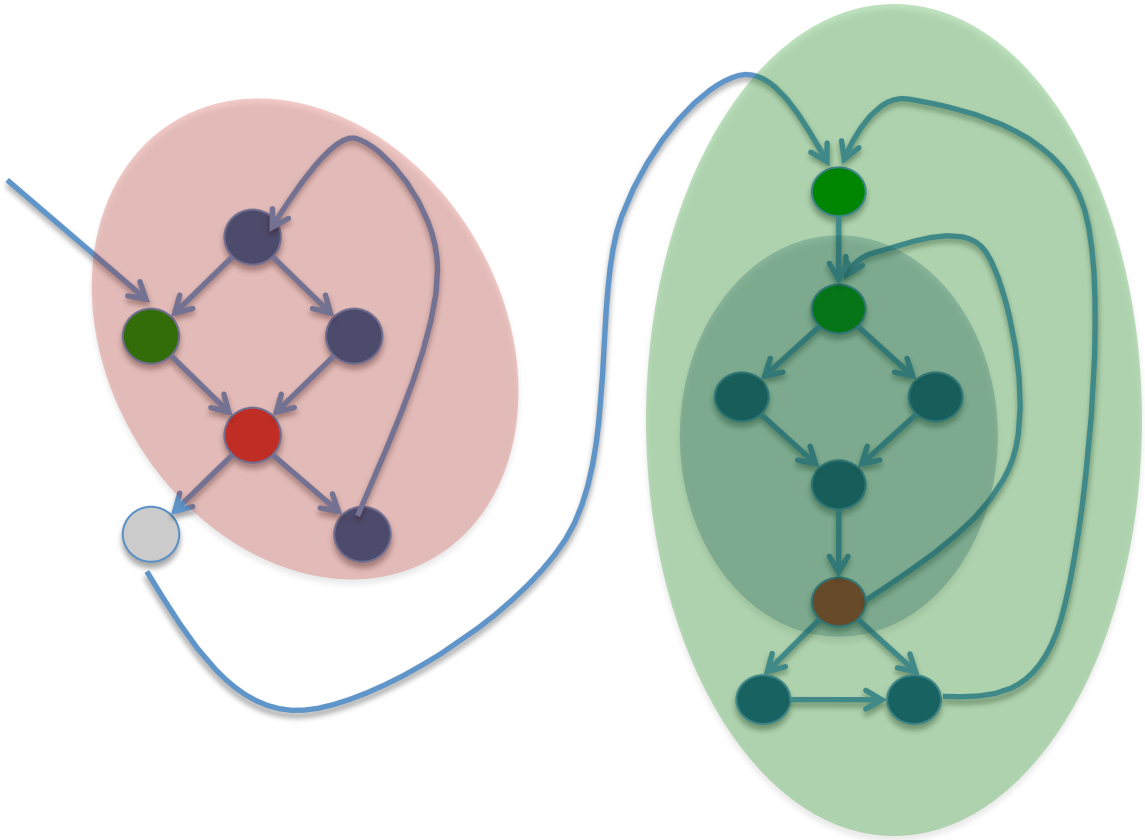
- Every node is reachable from the header & the header is reachable from every node.
  - A loop is a *strongly connected component*

- No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes

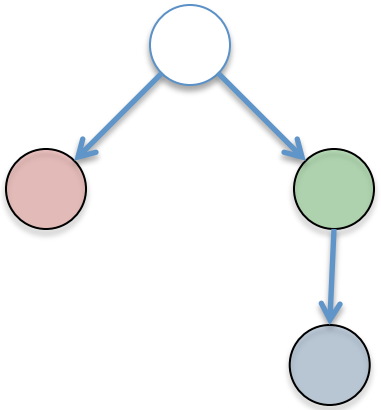


Nested Loops

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



Control Tree:



The control tree depicts the nesting structure of the program.

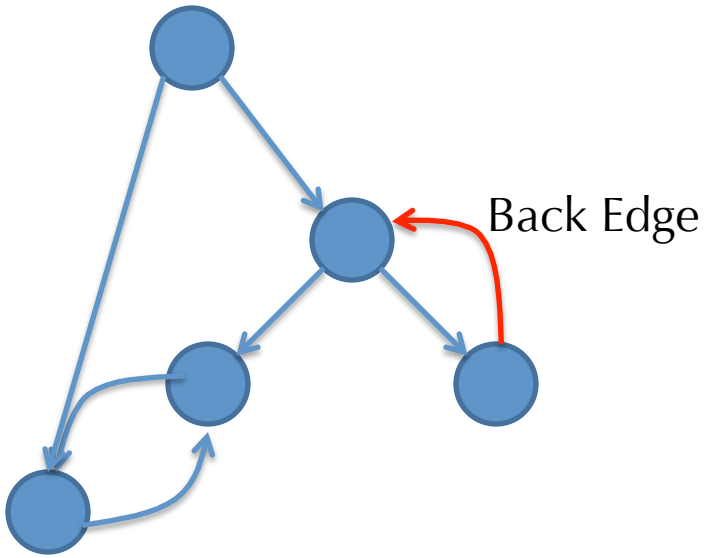
Control-flow Analysis

- 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.

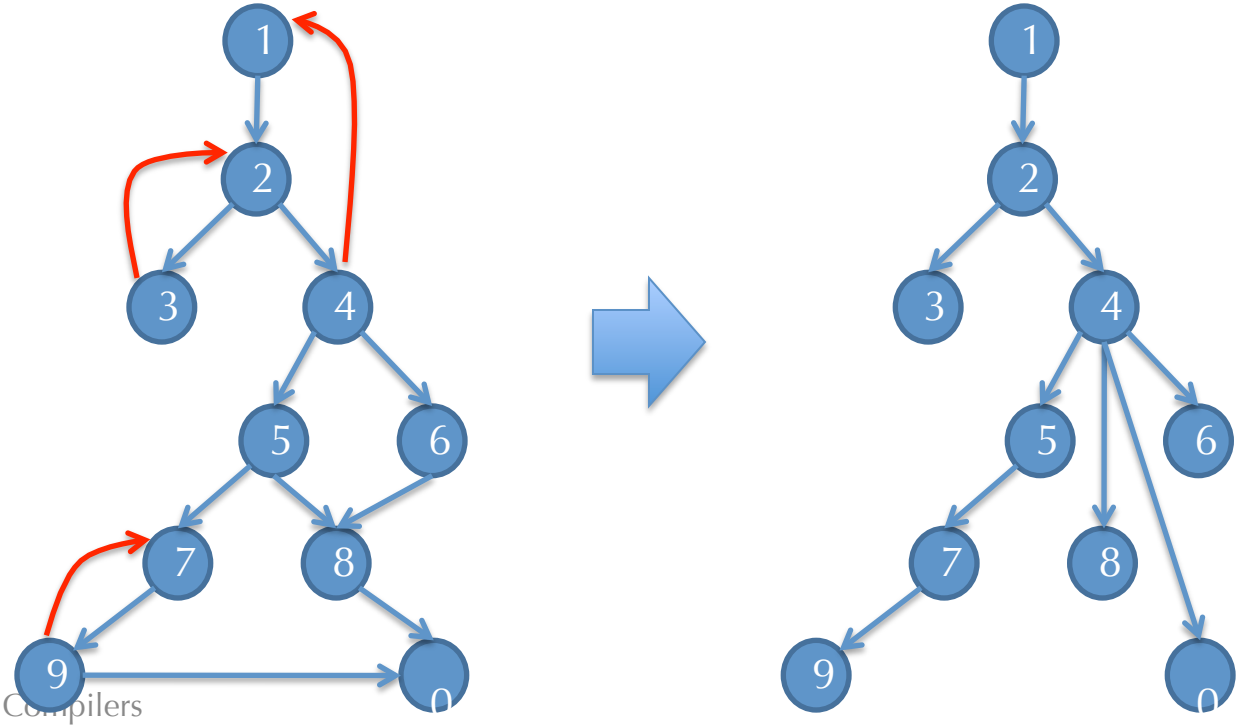
- An edge in the graph is a *back edge* if the target node dominates the source node.

- 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

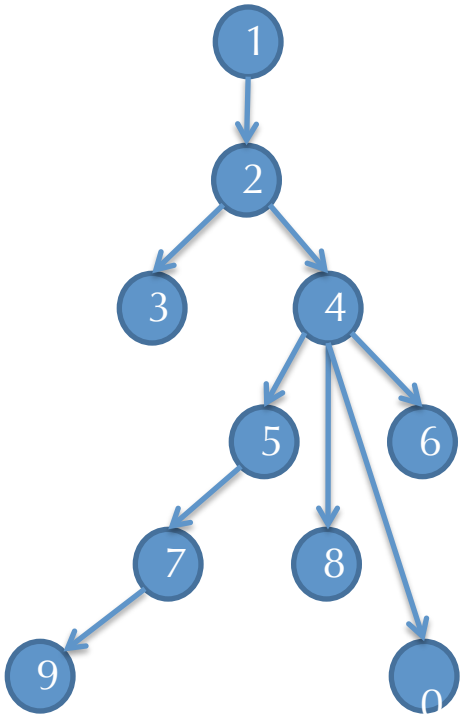
- We can define Dom[n] as a forward dataflow analysis.
  - Using the framework we saw earlier: Dom[n] = out[n] where:
- "A node B is dominated by another node A if A dominates *all* of the predecessors of B."
  - $in[n] := \bigcap_{n' \in pred[n]} out[n']$
- "Every node dominates itself."
  - $out[n] := in[n] \cup \{n\}$

- Formally: $\mathcal{L}$ = set of nodes ordered by $\subseteq$
  - $T = \{\text{all nodes}\}$
  - $F_n(x) = x \cup \{n\}$
  - $\sqcap$ 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

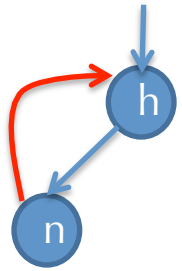
- More efficient way to represent Dom sets is to store the dominator *tree*.
  - doms[b] = immediate dominator of b
  - doms[8] = 4, doms[7] = 5
- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
  - Traverse up tree, looking for least common ancestor: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



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

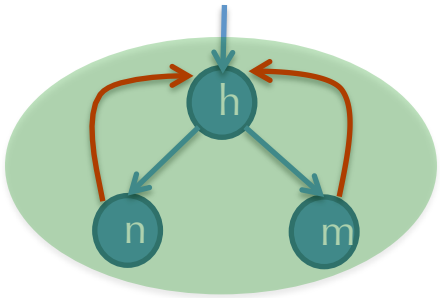
Completing Control-flow Analysis

- 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



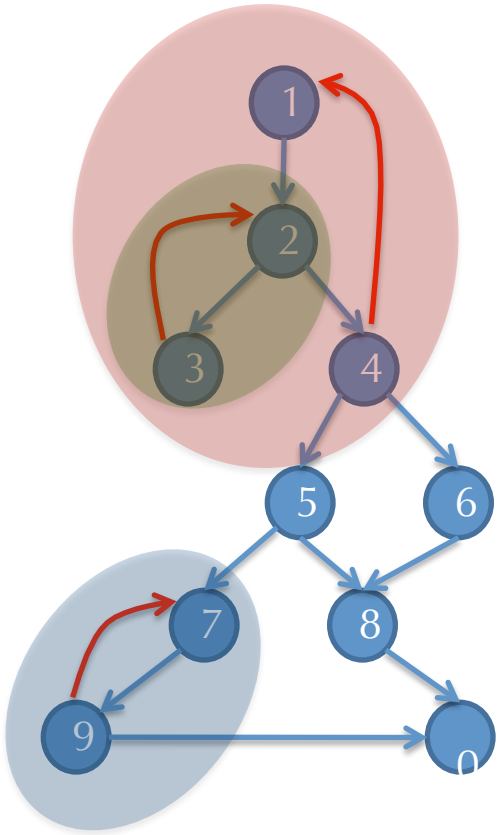
- For each back edge n → h, find the natural loop:
  - {n′ | n is reachable from n′ in G – {h}} ∪ {h}

- Two loops may share the same header: merge them



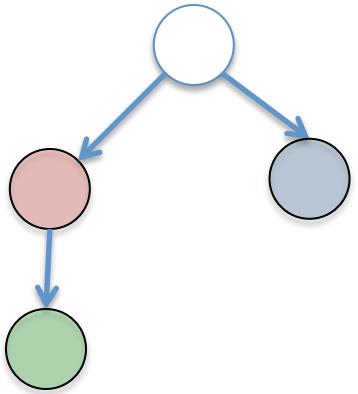
- Nesting structure of loops is determined by set inclusion
  - Can be used to build the control tree

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 i32
%y = alloca i32
store i32* %x, 3
store i32* %y, 0
%x1 = load %i32* %x
%tmp1 = add i32 %x1, 1
store i32* %x, %tmp1
%x2 = load %i32* %x
%tmp2 = add i32 %x2, 2
store i32* %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 i32 3, 0
%y1 = add i32 0, 0
%x2 = add i32 %x1, 1
%y2 = add i32 %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 i32 %y1, 1
  br label %merge
else:
  %x3 = mult i32 %y1, 2
merge:
  %z2 = %add i32 ???, 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 i32 %y1, 1
  br label %merge
else:
  %x3 = mult i32 %y1, 2
merge:
  %x4 = phi i32 %x2, %then, %x3, %else
  %z2 = %add i32 %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 (discussed soon)

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

body:
  %x2 = phi i32 %x1, %entry, %x3, %body
  %x3 = add i32 %x2, %y1
  %p = icmp slt i32, %x3, 10
  br i1 %p, label %body, label %after

after:
  …
```