Control Dependencies

- A node (basic block) \( Y \) is control-dependent on another \( X \) if \( X \) determines whether \( Y \) executes
  - there exists a path from \( X \) to \( Y \) s.t. every node in the path other than \( X \) and \( Y \) is post-dominated by \( Y \)
  - \( X \) is not post-dominated by \( Y \)

Example

Control Dependence Graph

- Control dependence graph: \( Y \) descendent of \( X \) iff \( Y \) is control dependent on \( X \)
  - label each child edge with required condition
  - group all children with same condition under region node

- Program dependence graph: super-impose dataflow graph (in SSA form or not) on top of the control dependence graph
Another example

Summary of Control Dependence Graph

• More flexible way of representing control dependencies than CFG (less constraining)

• Makes code motion a local transformation

• However, much harder to convert back to an executable form

Course summary so far

• Dataflow analysis
  – flow functions, lattice theoretic framework, optimistic iterative analysis, precision, MOP

• Advanced Program Representations
  – SSA, CDG, PDG

• Along the way, several analyses and opts
  – reaching defns, const prop & folding, available exprs & CSE, liveness & DAE, loop invariant code motion

• Next: dealing with procedures

Interprocedural analyses and optimizations
Costs of procedure calls

- Up until now, we treated calls conservatively:
  - make the flow function for call nodes return top
  - start iterative analysis with incoming edge of the CFG set to top
  - This leads to less precise results: “lost-precision” cost

- Calls also incur a direct runtime cost
  - cost of call, return, argument & result passing, stack frame maintainance
  - “direct runtime” cost

Addressing costs of procedure calls

- Technique 1: try to get rid of calls, using inlining and other techniques
- Technique 2: interprocedural analysis, for calls that are left

Inlining

- Replace call with body of callee
- Turn parameter- and result-passing into assignments
  - do copy prop to eliminate copies
- Manage variable scoping correctly
  - rename variables where appropriate

Program representation for inlining

- Call graph
  - nodes are procedures
  - edges are calls, labelled by invocation counts/frequency
- Hard cases for building call graph
  - calls to/from external routines
  - calls through pointers, function values, messages
- Where in the compiler should inlining be performed?

Inlining pros and cons (discussion)

- Pros
  - eliminate overhead of call/return sequence
  - eliminate overhead of passing args & returning results
  - can optimize callee in context of caller and vice versa
- Cons
  - can increase compiled code space requirements
  - can slow down compilation
  - recursion?
- Virtual inlining: simulate inlining during analysis of caller, but don’t actually perform the inlining
Which calls to inline (discussion)

- What affects the decision as to which calls to inline?

Which calls to inline

- What affects the decision as to which calls to inline?
  - size of caller and callee (easy to compute size before inlining, but what about size after inlining?)
  - frequency of call (static estimates or dynamic profiles)
  - call sites where callee benefits most from optimization (not clear how to quantify)
  - programmer annotations (if so, annotate procedure or call site? Also, should the compiler really listen to the programmer?)