1 Dominator dataflow analysis

See slides from previous lecture for the dataflow analysis that computes the set of dominators for each node. The key is to note that if node A dominates node B, it must dominate all predecessors of B. This leads straightforwardly to a forward dataflow analysis. The same analysis, if run on the reversed CFG, computes the *postdominators* for each node: nodes that must be encountered on any path from the node to an exit node.

2 Finding loops

To find loops, we look for back edges $n \to h$ in the CFG that go from a node n to a node h that dominates it. Each loop has at least one back edge. To find the rest of the nodes in the loop, we do a depth-first traversal in the reversed CFG, starting from n and stopping when we hit h. Every node we reach en route to h must be in the loop: it is reachable from h and reaches n, so it is in the same strongly connected component, and further, since it reaches n and is reachable by h, it is dominated by h (if not, h wouldn't dominate n either).

Given any two loops defined by this technique, either they are disjoint, nested, or share the same header node. We merge loops that share the same header node and consider them to be a single loop; the result is a set of loops that are either disjoint or nested, and therefore form a *control tree*. We are now ready to do some loop optimizations.

3 Loop-invariant code motion

Loop-invariant code motion is an optimization in which computations are moved out of loops, making them less expensive. The first step is to identify *loop-invariant expressions* that take the same value every time they are computed.

An expression is loop-invariant if

1. It contains no memory operands that could be affected during the execution of the loop (i.e., that do not alias any memory operands updated during the loop). To be conservative, we could simply not allow memory operands at all, though fetching array lengths is a good example of a loop-invariant computation that can be profitably hoisted before the loop.
2. And, the definitions it uses (in the sense of reaching definitions) either come from outside the loop, or come from inside the loop but are loop-invariant themselves.

The recursive nature of this definition suggests that we should use an iterative algorithm to find the loop-invariant expressions, as a fixed point. The algorithm works as follows:

1. Run a reaching definitions analysis.
2. Initialize INV := {all expressions in loop}.
3. Repeat until no change:
   - Remove all expressions from INV that use variables x with more than one definition with at least one definition inside the loop, or whose single definition $x = e$ in the loop has $e \notin INV$.

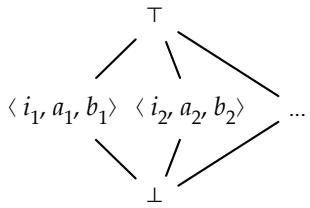

Figure 1: Dataflow values for induction variables analysis

4 Code transformations

There are actually two versions of loop-invariant code motion. One involves hoisting a computation before the loop, the other hoists the actual assignment to the variable used. We can move the assignment $x = e$ before the loop header if (1) it dominates all loop exits and is the the only definition of x in the loop, and (2) it is the only definition of x reaching uses of x in the loop.

If these conditions are not satisfied, we can still hoist a loop-invariant expression e out of the loop and assign it to a new variable t. Then the original assignment $x = e$ is changed to $x = t$.

5 Induction variables

A variable v is called an *induction variable* if it takes on the following values during the loop for some constants c and d: the values $ci + d$ for $i = 0, 1, 2, \ldots$.

If an induction variable v takes on a new value for each loop iteration, so that i is the number of the loop iteration, then v is called a *linear induction variable*.

Induction variables are easier to reason about than other variables, enabling various optimizations. Linear combinations of induction variables result in induction variables, which means that often loops have several induction variables that are related to each other.

A *basic induction variable* i is one whose only definition in the loop has the form $i = i + c$ for some loop-invariant value c (usually a constant). A basic induction variable is linear if its definition either dominates or postdominates every other node in the loop.

A *derived induction variable* j is a variable that is defined in terms of another induction variable. It can be expressed as $ci + d$ where i is a basic induction variable. All the derived induction variables that are based on the same basic induction variable i are said to be in the same *family*.

We write $\langle i, a, b\rangle$ to denote a derived induction variable in the family of basic induction variable i, with the formula $ai + b$. A basic induction variable i can therefore be written in this notation as $\langle i, 1, 0\rangle$.

For example, in the following code i is a basic induction variable, j is a linear basic induction variable, and k and l are derived induction variables in the same family as i:

```
while (i < 10) {
    j = j + 2;
    if (j > 4) i = i + 1;
    k = j + 10;
    l = k * 4;
}
```

6 Finding induction variables

We can find induction variables with a dataflow analysis over the loop. The domain of the analysis is mappings from variable names to the lattice of values depicted in Figure 1. In this lattice, two induction variables are related only if they are the same induction variables; a variable can also be mapped to $\bot$,

meaning that it is not an induction variable, or $\top$, meaning that no assignments to the variable have been seen so far, and hence it is not known whether it is an induction variable.

A value computed for a program point is a mapping from variables to the values above; that is, a function. The ordering on these function is pointwise. To compute the meet of two such functions, we compute the meet of the two functions everywhere. In other words, if functions F_1 and F_2 map variable v to values l_1 and l_2 respectively, then $F_1 \sqcap F_2$ maps v to $l_1 \sqcap l_2$. This sounds complicated but is just the obvious thing to do.

To start the dataflow analysis, we first find all basic induction variables, which is straightforward. Then the initial dataflow value for each node is the function that maps all basic induction variables i to $\langle i, 1, 0\rangle$, and maps all other variables to $\top$.

The transfer functions for program nodes involve simple algebraic manipulations. For an assignment $k = j + c$ where j is an induction variable $\langle i, a, b\rangle$ and c is loop-invariant, we conclude $k \mapsto \langle i, a, b + c\rangle$. For a corresponding assignment $k = j * c$, we conclude $k \mapsto \langle i, ac, bc\rangle$. For other assignments $k = e$, we set $k \mapsto \bot$. Other variables are unaffected.