1 Postdominators

Previously we saw a 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 transposed CFG in which all the arrows are reversed, computes the *postdominators* for each node: nodes that must be encountered on any path from the node to an exit node.

2 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.

2.1 Analysis

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, including subexpressions}.
3. Repeat until no change:
   - Remove all expressions from INV that use variables x with more than one definition inside the loop, or whose single definition $x = e$ in the loop has $e \notin INV$.

2.2 Code transformation

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 $x = e$ with loop-invariant expression e before the loop header if:

1. it is the only definition of x in the loop,
2. it dominates all loop exits where x is live-out, and
3. it is the only definition of x reaching uses of x in the loop: it is not live-in at the loop header.

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

In either case we need to watch out for expressions e that might generate an exception or other error, because hoisting them ensures they are evaluated, even though they might not be evaluated in the original execution.

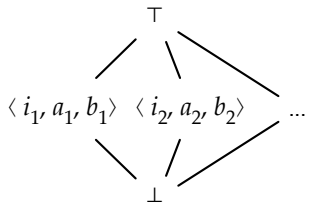

Figure 1: Dataflow values for induction variables analysis

3 Induction variables

A variable v is called an *induction variable* for a loop if it is updated during loop only by adding some loop-invariant expression to it.

If an induction variable v takes on at loop iteration n a value of the form $cn + d$, where c and d are loop-invariant, then v is a *linear induction variable*.

Induction variables are easier to reason about than other variables, enabling various optimizations. Linear functions of induction variables are also 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 definitions in the loop are equivalent to $i = i + c$ for some loop-invariant expression c (typically a constant). The value c need not be the same at every definition. A basic induction variable is linear if it has a single definition and that definition either dominates or postdominates every other node in the loop.

A *derived induction variable* j is a variable that can be be expressed as $ci + d$ where i is a basic induction variable, and c, d are loop-invariant. All the derived induction variables that are based on the same basic induction variable i are said to be in the same *family* or *class*.

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 $\langle i, 1, 0 \rangle$.

In the following code, i is a basic induction variable, j is a linear basic induction variable, k and l are linear derived induction variables in the family of j, and m is a derived induction variable in the i family.

```
while (i < 10) {
    j = j + 2;
    if (j > 4) i = i + 1;
    i = i - 1;
    k = j + 10;
    l = k * 4;
    m = i * 8;
}
```

4 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.

5 Strength reduction

Consider the following loop, which updates a sequence of memory locations:

```
while (i < a.length) {
    j = a + 3*i;
    [j] = [j] + 1;
    i = i + 2;
}
```

The variable j is computed using multiplication, but it is a derived induction variable $\langle i, 3, a\rangle$ in the notation of the previous lecture, in the same family as the basic induction variable i.

The idea of strength reduction using induction variables is to compute j using addition instead of multiplication. Perhaps even more importantly, we will compute j without using i, possibly making i dead.

The optimization works as follows for a derived induction variable $\langle i, a, b\rangle$:

1. Create a new variable s initialized to $a * i + b$ before the loop.
2. Replace the definition $j = e$ with $j = s$.
3. After the assignment $i = i + c$, insert $j = j + ac$.

On our example above, this has the following effect:

```
s = a + 3*i;
while (i < 10) {
    j = s;
    [j] = [j] + 1;
    i = i + 2;
    s = s + 6;
}
```

6 Induction variable elimination

Once we have derived induction variables, we can often eliminate the basic induction variables they are derived from. After strength reduction, the only use of basic induction variables is often in the loop guard. Even this use can often be removed through *linear-function test replacement*, also known as removal of *almost-useless variables*.

If we have an induction variable whose only uses are being incremented ($i = i + c$) and for testing a loop guard ($i < n$ where n is loop-invariant), and there is a derived induction variable $k = \langle i, a, b\rangle$, we can write the test $i < n$ as $k < a * n + b$. With luck, the expression $a * n + b$ will be loop-invariant and can be hoisted out of the loop. Then, assuming i is not live at exit from the loop, it is not used for anything and its definition can be removed. The result of applying this optimization to our example is:

```
s = a + 3*i;
t = a + 3*a.length;
while (s < t) {
    j = s;
    [j] = [j] + 1;
    s = s + 6;
}
```

A round of copy propagation and dead code removal gives us tighter code:

```
s = a + 3*i;
t = a + 3*a.length;
while (s < t) {
    [s] = [s] + 1;
    s = s + 6;
}
```