1 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 post-dominates 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;
}
```

2 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.

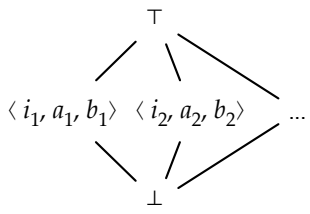

Figure 1: Dataflow values for induction variables analysis

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.

3 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;
}
```

4 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;
}
```

5 Bounds check removal

In type-safe languages, accesses to array elements generally incur a bounds check. Before accessing `a[i]`, the language implementation must ensure that `i` is a legal index. For example, in Java array indices start at zero, so the language must test that $0 \leq$ `i` $<$ `a.length`.

Returning to our example from earlier, after strength reduction we can expect the code to look more like the following (or the equivalent CFG):

```
s = a + 3*i;
while (i < a.length) {
    j = s;
    if (i < 0 | i ≥ a.length) goto L_err;
    [j] = [j] + 1;
    i = i + 2;
    s = s + 6;
}
```

This extra branch inside the loop is likely to add overhead. Furthermore, it prevents the induction variable elimination operation just discussed.

One simple improvement we can make is to implement the check $0 \leq$ `i` $<$ `n` in a single test. Assuming that `n` is a signed positive integer, and `i` is a signed integer, this test can be implemented by doing an *unsigned* comparison `i`$<$`n`. If `i` is negative, it will look like a large unsigned integer that will fail the unsigned comparison, as desired. Processor architectures have a unsigned comparison mode that supports this. For example, the `jae` instruction ("jump above or equal") on the Intel architecture implements unsigned comparison.

Even better would be to eliminate the test entirely. The key insight is that the loop guard (in this case, `i` $<$ `a.length`) often ensures that the bounds check succeeds. If this can be determined statically, the bounds check can be removed. If it can be tested dynamically, the loop can be split into two versions, a fast version that does not do the bounds check and a slow one that does.

The bounds-check elimination works under the following conditions:

1. Induction variable j has a test ($j < u$) vs. a loop-invariant expression u, where the loop is exited if the test failed.

2. Induction variable k in the same family as j has a test equivalent to $k < n$ vs. a loop-invariant expression n, where $j < u$ implies $k < n$, again exiting the loop on failure. The test on k must be dominated by the test on j, and k and j must go in the same direction (both increase or both decrease).

Under these conditions, the bound checks on k is superfluous and can be eliminated. But when does $j < u$ imply $k < n$? Suppose that $j = \langle i, a_j, b_j \rangle$ and $k = \langle i, a_k, b_k \rangle$. If the j test succeeds, then $a_j i + b_j < u$. Without loss of generality, assume $a_j > 0$. Then this implies that $i < (u - b_j)/a_j$. Therefore, $k = a_k i + b_k < a_k(u - b_j)/a_j + b_k$. If we can show statically or dynamically that this right-hand side is less than or equal to n, then we know $k < n$. So the goal is to show that $a_k(u - b_j)/a_j + b_k \leq n$. This can be done either at compile time or by hoisting a test before the loop. In our example, the test for `i` $<$ `a.length` is that $1 * (\texttt{a.length} - 0)/1 + 0 \leq \texttt{a.length}$, which can be determined statically. The compiler does still need to insert a test that $i \geq 0$ before the loop:

```
s = a + 3*i;
if (i < 0) goto L_err;
while (i < a.length) {
    j = s;
    [j] = [j] + 1;
    i = i + 2;
    s = s + 6;
}
```

After linear-function test replacement, this code example will become subject to induction variable elimination.