Last time we saw that loops can contain induction variables that take on a series of values $ai + b$ for constants a and b, and indices $i = 0, 1, 2, \ldots$. Assuming we have identified induction variables, there are several optimizations we can use to speed up loops.

1 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 deifnition $j = e$ 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;
}
```

2 Induction variable elimination

Once we have derived induction variables, we can often eliminate the basic induction variables they are derived from. If we have an induction variable whose only uses are being incremented ($i = i + c$) and for testing a loop guard ($i \langle n$ where n is loop-invariant), and there is a derived induction variable $k = \langle i, a, b\rangle$, we can write the test $i \langle n$ as $k \langle a * n + b$. 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;
}
```

3 Bounds check removal

In type-safe languages, accesses to array elements generally incur a bounds checks. 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 (more likely, 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, again exiting the loop on failure. The test on k is dominated by the test on j, and k and j go in the same direction (both increase or both decrease).

Under these conditions, and if $j < u$ implies $k < n$, the bound checks on k is superfluous and can be eliminated. 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 $1 * (\texttt{a.length} - 0)/1 + 0 \leq$ `a.length`, which can be determined statically. The compiler does still need to insert a test that $i \geq 0$, but then this example becomes subject to induction variable elimination:

```
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;
}
```

4 Loop unrolling

Loop guards and induction variable updates add significant overhead to short loops. The cost of loop guards involving induction variables can often be reduced by *unrolling* loops to form multiple consecutive copies. Multiple loop guard tests can be then be combined into a single conservative test. If the loop is unrolled to make n copies, and the loop guard has the form $i < u$ where u is a loop-invariant expression and i is an induction variable with stride c that is updated at most once per loop iteration, the test $i < c(n-1)$ conservatively ensures that all n loop copies can be run without having any guard fail.

Since this guard is conservative, there still may be $< n$ iterations left to be performed. So the unrolled loop has to be followed by another copy of the original loop, the *loop epilogue*, which performs any remaining iterations one by one. Since loop unrolling therefore results in $n + 1$ copies of the original loop, it trades off code size for speed. If the original loop was only going to be executed for a small number of iterations, loop unrolling could hurt performance rather than improve it.

Updates to basic linear induction variables inside the unrolled loop can also be combined. If a variable i is updated with the statement $i = i + c$, then n copies of the update can be replaced with a single update $i = i + nc$. However, any uses of i within the copies of the loop must be changed as well – in the second loop copy, to $i + c$, in the third copy (if any), to $i + 2c$, and so on. These additions can often be folded into existing arithmetic computations.