Lecture Notes on Induction Variables

15-411: Compiler Design
André Platzer

Lecture 18

1 Introduction

More information can be found in [App98, Ch 18.1-18.3] and [Muc97].

Last lecture, we have seen strength reduction. In order to perform strength reduction, however, we need to know which of the variables change linearly in the loop. These are called induction variables.

2 Induction Variables

Once we have identified a loop (e.g., natural loop), one of the central questions about it is, which variables are induction variables of the loop. In for-loops, there are syntactical indicators if the step is of the form `i++`. But that alone does not make i a proper induction variable, because there could be further assignments to i. Furthermore, other variables could effectively be induction variables, even if they are not written in the step part of a for-loop. Finally, induction variables are also of interest for while or repeat-until loops, where they are not identified syntactically. Consequently, we need an analysis to identify induction variables.

Generally, we will consider variables to be induction variables if their value is linear in the number of loop iterations. We call a variable i a *basic induction variable* if the only assignments to i in the loop body are of the form $i = i + c$ (or $i = i - c$) for a loop-invariant expression c. We call variable j a *derived induction variable* if it only assumes values of the form $j = a_j * i + b_j$ for a basic induction variable i and loop-invariant expressions a_j, b_j. Of course, it is again undecidable whether a variable only assumes those values, so we will confine ourselves to just finding some cases where

```
s = 0
i = 0
l1: if (i ≥ n) goto l2
j = 4*i              // derived j=4*i+0
k = j + a            // derived k=1*j+a=(1*4)*i+a
x = M(k)
s = s + x
i = i + 1            // basic
goto l1
l2:
```

Figure 1: Example computing sum of 32bit array contents

we can show that i is a basic induction variable or a derived induction variable, respectively.

In fact, both basic and derived induction variables are linear functions of the basic induction variable. All of them have the form $j = a_j * i + b_j$ (where $i = 1 * i + 0$ is a special case).

There are several ways to find a derived induction variable j in a loop. We show one way following [App98]. Variable j is a derived induction variable if it is only defined once in the loop body with a definition $j = a_j * k + b_j$ for a (derived or basic) induction variable k and loop-invariant expressions a_j, b_j. If this variable k is a derived induction variable then we also require that its (unique) definition in the loop body is the only definition of k reaching the definition of j and that the corresponding basic induction variable i for k is not redefined on any path between the respective definitions of j and k.

$$\frac{\begin{array}{l} l : j = i \pm c \\ \neg inv(c) \end{array}}{\neg IV(j)}\; BIV_1 \qquad \frac{\begin{array}{l} l : j = i \pm c \\ \neg IV(i) \end{array}}{\neg IV(j)}\; BIV_2$$

$$\frac{\begin{array}{l} l : j = a_j * i \pm b_j \\ \neg inv(a_j) \vee \neg inv(b_j) \end{array}}{\neg IV(j)}\; IV_1 \qquad \frac{\begin{array}{l} l : j = a_j * i \pm b_j \\ \neg IV(i) \end{array}}{\neg IV(j)}\; IV_2$$

$$\frac{\begin{array}{l} l : j = \Phi(i_1, \ldots, i_n) \\ \neg inv(i_k) \\ \neg IV(i_k) \end{array}}{\neg IV(j)} \; \Phi IV$$

For IV_1 and IV_2, the case where a_j does not appear (corresponding to $a_j = 1$) actually includes BIV_1 and BIV_2 as a special case. But only the latter would be called basic induction variables. There also is a rule that if a variable i changes in any other way (e.g., $i = x * y$ or $i = f(x, y)$), then it is not an induction variable. We do not write this one down explicitly, because induction variable analysis is usually only pursued for variables that only change by linear and Φ functions anyhow.

For computing induction variables, we proceed as follows. We first just assume that all variables were induction variables. Then we successively throw candidates out that do not match the conditions. That is, we saturate the list of induction variables by saturating the database according to the rules above. For SSA programs, this is particularly easy.

```
S = set of all variables
repeat until fixedpoint:
  remove j from S if j not computed as one of the forms
    basic:
       j = i ± c for an $i \in S$ and a loop-invariant c
    derived:
       j = $a_j*i$ ± $b_j$ for an $i \in S$ and loop-invariant $a_j, b_j$
    flow:
      j = $\Phi(i_1, \ldots, i_n)$ and each $i_k$ loop-invariant or ∈ $S$
```

But on SSA, it turns out that induction variables are associated with a strongly connected component beginning with a Φ-function for that variable. Hence, implementations often first compute strongly connected components (subgraphs from which every node can reach every other) by Tarjan's algorithm and then consider one strongly connected component at a time.

3 Strength Reduction for Induction Variables

If we have found a basic induction variable i that is initialized to i_0 before the loop and a derived induction variable j, then we can replace j by a new induction variable j' as follows. Then we replace the loop

```
i = $i_0$
while (e) {
  ...
  j = $a_j$*i + $b_j$
  ... j ...
  i = i+c
  ... j ...
}
```

according to the strength reduction optimization by

```
i = $i_0$
j' = $a_j * i_0$ + $b_j$
while (e) {
  ...
  j = j'          // $j$ updates by shadow $j'$
  ... j ...
  i = i+c
  j' = j' + $a_j$*$c$  // increment $j'$ at every change of $i$
  ... j ...
}
```

After every assignment to the basic induction variable i, we increment the new variable j'. And the (single) assignment $j = a_j * i + b_j$ gets replaced by $j = j'$. Note that $a_j * c$ can either be computed by constant folding or is loop-invariant and can be moved outside. Finally, we can rely on copy propagation to optimize j away as much as possible. We can also use reassociation and constant folding to accumulate successive increments of j' within the loop body into one assignment if that is permitted.

Figure 2 on p 5 shows the result of strength reduction optimization of Fig. 1 on p 2. Note that the variable j' is quite useless, because its only purpose has become to assign to itself. This is what the neededness analysis from lecture 5 on dataflow analysis can figure out and eliminate j'. The only use of j' is to define itself and it's dead after the loop too.

4 Almost Useless Variables

If the induction variable i is still used in the loop body or loop test e then the assignments to i can either be kept, or, instead, uses of i can be recomputed from j and replaced by $(j - b_j)$ div a_j. The latter really only makes sense

```
s = 0
i = 0
j' = 0                  // j' not needed => dead
k' = a
l1: if (i ≥ n) goto l2
j = j'                  // dead
k = k'
x = M(k)
s = s + x
i = i + 1
j' = j' + 4             // not needed
k' = k' + 4
goto l1
l2:
```

Neededness analysis removes useless j'. Copy propagation of $k = k'$ gives

```
s = 0
i = 0
k' = a
l1: if (i ≥ n) goto l2
x = M(k')
s = s + x
i = i + 1              // almost useless
k' = k' + 4
goto l1
l2:
```

Figure 2: Example from Figure 2 after strength reduction for j and k.

when this division can be simplified arithmetically. At least we know that j changes in multiples of a_j.

In Figure 2 (bottom) there is an almost useless variable i. The reasoning is by using that k is derived from j by $k = a_k * j + b_k$, which is derived from i by $j = a_j * i + b_j$. Consequently,

$$i = (j - b_j)\ div\ a_j = (((k - b_k)\ div\ a_k) - b_j)\ div\ a_j$$

Thus, $i \geq n$ is equivalent to

$$(((k - b_k)\ div\ a_k) - b_j)\ div\ a_j \geq n$$

Inserting the relations from Fig. 1, we get

$$(((k - a)\ div\ 1) - 0)\ div\ 4 \geq n$$

i.e.

$$(k - a)\ div\ 4 \geq n$$

this is equivalent to the following, because we know that k will only change in multiples of its linear factor 4

$$k - a \geq 4 * n$$

i.e.,

$$k \geq 4 * n + a$$

This $4 * n + a$ is a loop-invariant expression that can be computed before the loop.

After optimizing the almost useless variable i away, we get Figure 3.

Remaining issues to deal with are showing that the transformed loop exit check will not decide different than the original code due to arithmetic overflows. This hinges on the fact that modular integer arithmetic defines a ring, but in the presence of division it is a bit more involved. The other issue is that the transformations above would have flipped the direction from $\geq$ to $\leq$ if $a_j < 0$, and would have flipped once more if $a_k < 0$. For constants, this is easy to decide. For loop invariant expressions, this can sometimes be decided at compile time and sometimes not. The general way to go is to compile the code for both comparisons and check the signs before the loop to either jump into the loop that uses $\geq$ comparison for exit or to the code that uses $\leq$ comparison. We will come back to this issue in a later lecture.

```
s = 0
k' = a
e = 4*n + a
l1: if (k' ≥ e) goto l2
x = M(k')
s = s + x
k' = k' + 4
goto l1
l2:
```

Figure 3: Strength reduced example from Figure 2 after eliminating the almost useless variable i.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.